Izrada društvene igre za više igrača u programskom alatu Unity

Žonja, Ian

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ian Žonja

IZRADA DRUŠTVENE IGRE ZA VIŠE IGRAČA U PROGRAMSKOM ALATU UNITY

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ian Žonja

Matični broj: 46439/17-R

Studij: *Informacijsko i programsko inženjerstvo*

IZRADA DRUŠTVENE IGRE ZA VIŠE IGRAČA U PROGRAMSKOM ALATU UNITY

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2021.

Ian Žonja

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema rada je izrada igre za više igrača u programskom alatu Unity. Unity je sustav s kojim možemo proizvesti video igre. Ipak, metodologija razvoja igre za jednog igrača i za više igrača nije ista. Ovdje se glavni naglasak stavlja na aspekt igranja s drugim igračima. Naučiti ćemo na koji način napraviti igru, kako ju testirati te koji su nam sve resursi potrebni da se napravi igra za više igrača, s korisničkim iskustvom visoke kvalitete.

Prvo ćemo u uvodu napisati nešto više o vlastitim motivacijama za pisanjem ove teme. Zatim ćemo proći glavne komponente koje je potrebno savladati da bi krenuli sa razvojem igre. U trećem poglavlju ćemo naučiti što je to Unity te kako započeti razvoj igre uz ovaj programski alat. Finalno ćemo načiti kako implementirati konkretne značajke potrebne u igri za više igrača.

Ključne riječi: Unity, razvoj, video igre, više igrača, klijent, server

Sadržaj

1. Uvod

Zanimanje za ovom temom dobio sam na prvoj godini fakulteta kada sam krenuo učiti programirati. Razvoj video igrača počeo je sa jezikom C++ gdje sam počeo sa vrlo jednostavnim igrama poput Križić/kružić, Zmija, Potapljanje brodova i druge. Izrada video igara uvelike mi je pomogla u stjecanju različitih iskustva koje danas posjedujem.

Za završni rad napravio sam prvu video igru za više igrača, u programskom alatu C++. Odmah sam znao da ću napraviti još jednu i za diplomski rad te je iz istog razloga nastao ovaj rad.

Igre za više igrača su prije svega jako zanimljive. Sve aktivnosti se događaju u realnom vremenu kako bi igre bile interaktivne. Sami razvoj ovakvih igara nije jednostavan. Štoviše, kada se razvija igra za više igrača, treba ju i testirati kao da ju igra više igrača, što zna biti itekako velik problem.

U ovoj temi baviti ćemo se Unity programskim alatom za izradu video igara. U temi se bavimo razvojem za više igrača pa ćemo prvo spomenuti vrste igara za više igrača. Objasniti ćemo mrežne protokole na kojima se igre razvijaju te zašto se oni koriste. Potom ćemo objasniti što je to Unity te ćemo proći glavne koncepte unutar Unitya koje je potrebno savladati kako bi se krenula razvijati igra za više igrača. Pričati ćemo o tome što su scene, preglednici unutar alata. Objasniti ćemo kako se kreiraju objekti te kako ih se pozicionira na scenu. Objasniti ćemo što su to objekti igre, komponente te objekti korisničkog iskustva. Također ćemo proći dinamičko instanciranje objekata te pozicioniranje klonova objekata na sceni samostalno te kao listu. Pred kraj rada ćemo reći nešto više o ostalim resursima koji su potrebni za razvoj igre za više igrača te finalno ponuditi zaključak.

2. Glavne komponente igara za više igrača

2.1. Mrežni protokoli

Sama industrija video igara postala je značajna pojavom interneta. Kao i ostatak interneta, video igre za više igrača po pravilu koriste mrežne protokole kao sredstvo razmjene podataka, posebice u realnom vremenu.

2.1.1. TCP/IP

„U svibnju 1974. Vint Cerf i Bob Kahn opisali su protokol umrežavanja za dijeljenje resursa pomoću paketnog prebacivanja između čvorova mreže.“ [1]. Sa ovim projektom se po prvi puta pokazala razmjena podataka na mreži, nalik internetu. Projekt je potom izrastao u mrežni protokol kakav se danas koristi, pod nazivom *Transmission Central Protocol.*

„TCP djeluje na način da prihvaća podatke na mreži, dijeli ih na komade i dodaje TCP zaglavlje stvarajući TCP segment. TCP segment se zatim enkapsulira u datagram Internet protokola (IP) i razmjenjuje s ostalim sudionicima na mreži.“[2].

Prednosti TCP protokola su sigurnost podataka. Naime, ovaj protokol osigurava da će se paket poslati primatelju. Paketi se mogu izgubiti no kako se radi o TCP segmentima, protokol će uključiti mehanizam ponovnog zahtjeva te finalno podizanja zastavice ukoliko dođe do neuspjeha. U tom slučaju pošiljatelj zna da nije došlo do greške sa njegove strane. Ukoliko je sve pošlo po redu, kod razmjene će se dogoditi takozvano „rukovanje“. Pošiljatelj šalje paket koji će se uspješno dostaviti te će primatelj primiti ispravan odgovor.

2.1.2. UDP

UDP djeluje na način da informaciju od jednog sudionika na mreži podijeli na datagrame koji imaju informaciju o tome gdje trebaju stići. Ipak, mreža je nestabilna te se mnogi datagrami mogu izgubiti na putovanju.

Za razliku od TCP-a, ovdje se sudionici ne rukuju i zbog toga UDP nije dobar izbor kod mnogih procesa koji zahtijevaju da svaki podatak sigurno dođe do svog primatelja. Ipak, protokol izvršava razmjenu podataka brže što je jako bitno za industriju video igara gdje je važno dostaviti paket u realnom vremenu.

2.1.3. HTTP

Hypertext Transfer Protokol nastao je s ekspanzijom interneta kao odgovor na brojne probleme koji su postojali na mreži. Web stranice koriste HTTP protokol za razmjenu paketa na mreži te se protokol bazira na TCP arhitekturi.

Protokol djeluje na način da pošiljatelj šalje određeni zahtjev te očekuje odgovor na isti. Svaki zahtjev i odgovor imaju svoju šifru odgovora. Prema tome se zna o kakvom je zahtjevu i odgovoru riječ. HTTP protokol po standardu nalazi na portu 80, dok TCP protokol može biti na bilo kojem portu. Neke od vrsti zahtjeva su GET, POST, PUT, PATCH, DELETE. Također i od najčešćih odgovora su OK, Not Found, Bad Request...

HTTP protokol često nije najbolji izbor za razvoj igre jer je namijenjen za web aplikacije, odnosno aplikacije koje se pokreću na pregledniku. Usprkos tome, industrija video igara baziranih na web preglednicima je u konstantnom porastu, a trenutno vrijedi oko 26 milijardi dolara.

2.2. Transportni sloj video igara

Svaka video igre za više igrača imaju neki oblik međusobne razmjene podataka kako bi imali interakciju. Podaci se razmjenjuju pomoću jednog od nekolicine različitih tehnologija koje podržavaju ovakav oblik razmjene. Gotovo svi se baziraju na razmjeni podataka na mreži, osim jednoga.

2.2.1. Igra za više igrača van mreže

Igre van mreže moraju koristiti zajednički uređaj kako bi se podaci interaktivno razmjenjivali. Iz tog razloga se ovakav oblik igara ne proizvodi često. Kako bi se postiglo dobro korisničko iskustvo, najčešće igru treba podijeliti na maksimalno 4 igrača koji će imati svaki svoj dio na ekranu.

2.2.2. LAN igra za više igrača

Local Area Network je vrsta mreže koja povezuje računala na manjoj geografskoj udaljenosti te nije za širu upotrebu. Igrači se najčešće nalaze u istoj prostoriji kada igraju preko LAN-a. 90-ih godina je stekao popularnost u industriji video igara te mu je sa razvojem interneta popularnost danas pala. Trend lan igara doveo je i do subkulture ljudi koji su fanovi ovakve vrste druženja.

Usprkos geografskom limitu, LAN ima nekoliko važnih značajki za video igre. Podaci se razmjenjuju bez kašnjenja te je skalabilan. Razvoj je jednostavniji i gotovo besplatan.

2.2.3. P2P transportni sloj

Peer to peer sloj je vrsta transportnog sloja kod kojega se paketi na mreži razmjenjuju korištenjem mrežnog protokola direktno između pošiljatelja i primatelja, bez upotrebe centralnog servisa.

2.2.3.1. Direktni P2P

Kod direktnog sloja svi sudionici moraju međusobno razmijeniti poruku kako bi bili usklađeni u realnom vremenu. Zbog kompleksnosti razmjene se najčešće direktni peer to peer sloj koristi samo u igrama za dva igrača.

2.2.3.2. Klijent-server P2P

Klijent-server je način izvedbe kod kojega je jedan igrač ujedno i servis za igru. Prema tome, svi ostali sudionici komuniciraju samo sa tim igračem.

Prednost ovakve arhitekture je u tome što se omogućila skalabilnost razmjene informacija sa brojem igrača. Ipak, nedostatak je taj da je igrač ujedno i servis. Ukoliko su ostali igrači geografski previše udaljeni od igrača koji djeluje kao servis, kašnjenje paketa će biti preveliko te će igra izgubiti na svojoj interaktivnosti.

2.2.4.Klijent-Server putem dediciranog servisa

Najčešća arhitektura za razmjenu informacija na transportnom sloju je klijent-server arhitektura s dediciranim serverom. Kod ovog oblika klijenti se spajaju na server korištenjem nekog od prethodno navedenih mrežnih protokola. Server prima informaciju te se ponaša kao napredni servis za obradu informacija te nadzor.

Industrija koristi ovakvu implementaciju iz razloga što je razvoj jednostavan, a osigurava skalabilnost sa brojem igrača. Također, geografska lokacija dediciranog servera je poznata. Kako bi pokrivenost bile veća, može se proširiti mreža dediciranih servera koji pokrivaju različite geografske lokacije, što osigurava prijenos paketa u realnom vremenu.

2.3. Klijentske aplikacije

Korisnici video igara pokreću igre putem uređaja koji posjeduju. One se najčešće razvijaju za popularne konzole, računala ili mobitele, a manje i web preglednike.

Video igre se najčešće razvijaju korištenjem nekog od Game Enginea, poput Unity-a. Proizvođači najčešće prodaju ovu aplikaciju svojim korisnicima, ili je dijele besplatno uz neki

drugi oblik monetizacije. Instalacija aplikacije na uređaj mora biti jednostavna pa se danas transakcija i preuzimanje potrebnih resursa za instalaciju vrše korištenjem javnih biblioteka za igre.

2.4. Pozadinske aplikacije za nadzor i obradu podataka

Serveri imaju svoje servise koji se implementiraju korištenjem nekog od mnoštva programskih jezika. To su skripte koje obrađuju podatke te poslužuju korisnike. Ispravna implementacija nije jednostavna jer se servisi moraju pobrinuti da se prikaz podataka i odvijanje procesa igre na klijentskim aplikacijama, svim igračima odvija na ispravan način, u realnom vremenu. Tako se na primjer ove aplikacije moraju pobrinuti i da pridruživanje korisnika (igrača) igri prođe glatko i bez korisničkih poteškoća.

Osim same skalabilnosti igračkog iskustva, potrebno je i osigurati podatke na mreži. Posebice je to važno kod igara koje nude takozvanu trgovinu unutar igre, gdje se mogu izvršiti mikro transakcije za kupovinu. Podaci moraju biti zaštićeni te se neki standardizirani procesi moraju ispuniti. Pozadinske, „Backend“ aplikacije se brinu za sve sigurnosne aspekte, poput autentifikacije, autorizacije, i obradu transakcija.

Implementacija ovakvog sustava može biti jako kompleksna te skupa. Često proizvođači koriste puno alata i aplikacija od trećih strana koje pojednostavljuju implementaciju cijele arhitekture. Ipak i kao takvi, mnoštvo resursa ima svoju cijenu.

2.5. Konfiguracija sustava

Da bi se postigla interaktivnost u realnom vremenu visoke kvalitete, cijeli sustav mora biti pažljivo izgrađen. Svaki server mora imati jaki hardver kako bi podržavao mnoštvo klijenata. Za visoku geografsku pokrivenost, serveri se umnožavaju na različitim lokacijama te tvore zajedničku mrežu igrača.

Potrebno je pažljivo povezati igrače na sličnoj geografskoj lokaciji putem zajedničkog servera. U tom slučaju korisnici imanju kašnjenje paketa dovoljno manje, što ne utječe na igrivost. Konfiguracija servera je također kompleksan proces koji se najčešće zamjenjuju trećim stranama, koje nude usluge na oblaku (cloud).

3. Unity sustav za igre

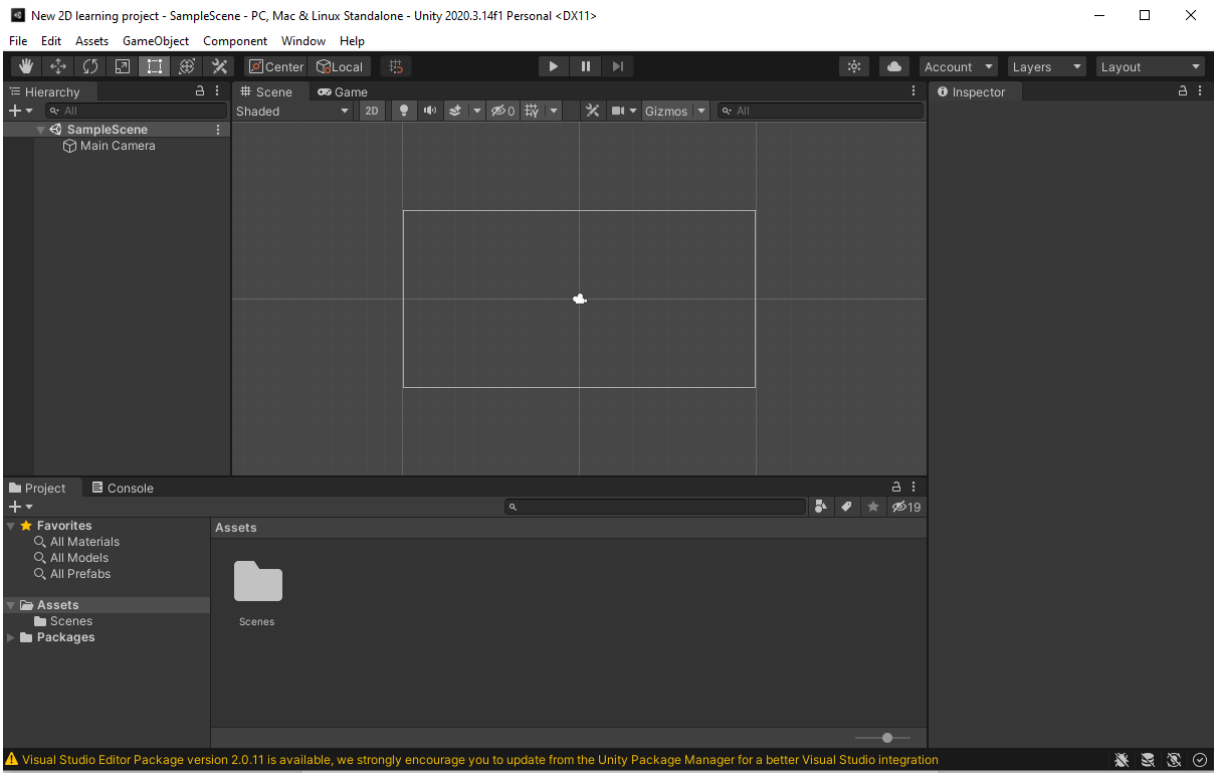
„Unity je sustav za igre na više platformi koji je razvila tvrtka Unity Technologies, koji je prvi put najavljen i objavljen u lipnju 2005. na svjetskoj konferenciji programera tvrtke Apple Inc. kao ekskluzivni pokretač igara za Mac OS X.“ [3].

Javnih sustava poput Unitya nema mnogo. Popularni su i Unreal Engine, Godot, GameMaker te nekolicina ostalih. Ovakvi sustavi uvelike ubrzavaju proces izrade video igara zbog svojih predefiniranih osobina. Radi njih se neke važne stvari poput pozicije objekata već matematički izračunate i programeri mogu svoje probleme rješavanjem proširivanjem preć preddefiniranih komponenti od strane sustava za igre. Neke kompanije razvijaju i vlastiti sustav za igre. Primjerice, popularna kompanija Rockstar Games razvija igrice na Rockstar Advanced Game Engine sustavu.

3.1. Scena

„Scene su mjesto na kojem radite sa sadržajem u cijelosti.“. Scene mogu sadržavati dio ili cijelu igru. „Na primjer, možete izgraditi jednostavnu igru u jednoj sceni, dok za složeniju igru možete koristiti jednu scenu po razini, svaku sa svojim okruženjima, likovima, preprekama, ukrasima i UI-jem. U projektu možete stvoriti bilo koji broj scena.“ [4].

Sve što se nalazi unutar scene se također nalazi i u hijerarhijskom pregledniku. Ukoliko kreiramo praznu scenu, na hijerarhijskom pregledniku neće biti ničega, osim kamere. Na sljedećoj slici možemo vidjeti kako izgleda prazna scena.



Slika 1: Prikaz scene u programskom alatu Unity

3.2. Hijerarhijski preglednik

Prozor za hijerarhiju se u pravilu nalazi na lijevoj strani, pored prikaza scene u Unityu. U ovom prozoru možemo vidjeti sve objekte koji su postavljeni na scenu. Jako je važan jer s njim možemo namještati hijerarhiju objekata. Drugim riječima putem ovog prozora na jednostavniji način pridružujemo elemente svojim roditeljima.

Slika 2: Prikaz hijerarhijskog preglednika unutar programskog alata Unity

3.3. Preglednik komponenata

Preglednik komponenata se često nalazi na desnoj strani Unity programa, unutar scene. Odabirom nekog od objekata sa hijerarhijskog prozora, ili spremljenih dodataka, preglednik će nam prikazati sve komponente i njihove atribute, koji se nalaze unutar objekta.

Glavni atributi komponenata se trebaju serijalizirati kako bi bile spremljene i prikazanekorisniku unutar preglednika komponenata. Kao takve mogu se i mijenjati, što programeru olakšava razvoj, posebice kod testiranja.

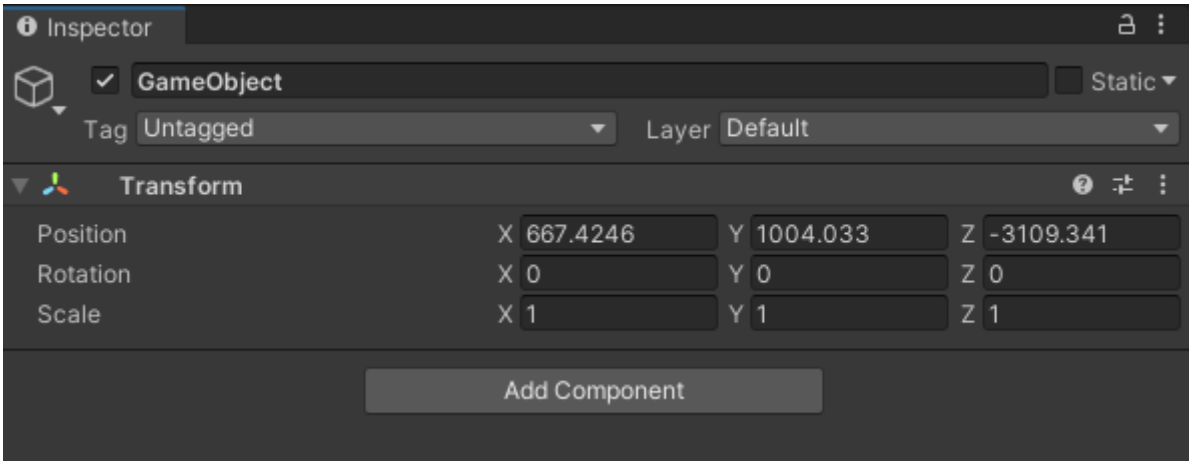
Slika 3: Prikaz preglednika komponenata unutar programskog alata Unity

3.4. Objekt igre i komponente

Objekt igre (eng. *Game object*) je temeljna klasa svih entiteta u scenama. Drugim riječima, sve objekte proširuje *GameObject* klasa. To je jako korisno jer se iz ovog objekta mogu izvući svi podaci o objektu.

Objektima igre je zajedničko to da sadržavaju komponente. Sve komponente koje objekt sadržava mu daju određene sposobnosti i pretvaraju ga u zaseban objekt sa svojom strukturom i funkcijom unutar igre. Dakle, podaci su spremljeni u komponentama.

Svi objekti imaju zajedničku komponentu pod nazivom *Transform*. Ova komponenta sadržava podatke o poziciji objekta unutar scene. Podaci su prikazani na pregledniku komponenata.

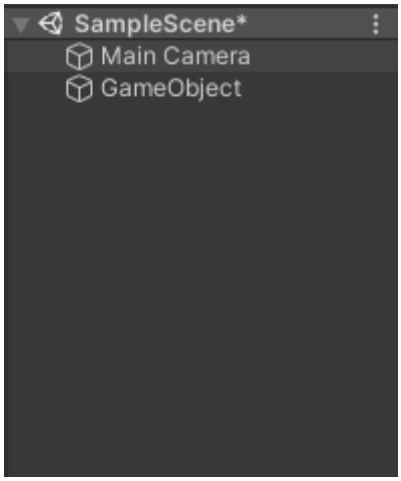


Slika 4: Prikaz Transform kompontente unutar preglednika komponenata

Objekt se kreira na dva načina:

- Unutar sustava za razvoj igre (eng. *game engine*)
- Programski

Ako želimo kreirati objekt unutar sustava za razvoj igre, dovoljno je desnim klikom na hijerarhijskom pregledniku odabrati opciju za kreiranje praznog objekta (eng. *Create empty*). Tada će se pojaviti na hijerarhijskom pregledniku pojaviti novi prazni objekt s predefiniranim imenom *GameObject.*



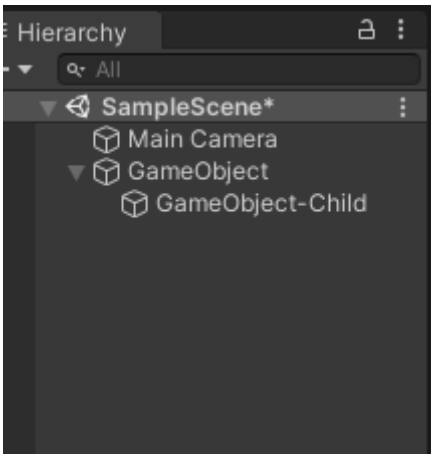
Slika 5: Prikaz praznog objekta igre unutar hijerarhijskog preglednika

Često moramo objekt kreirati u tokom izvršavanja programa. Takve objekte instanciramo programski. Kako bi to uspjeli, potrebno je implementirati skriptu koja generira novi objekt igre. Tada tu skriptu treba pridružiti nekom drugom objektu kao komponentu.

3.5. Hijerarhija objekata igre

Hijerarhija objekata je izrazito važna u Unityu jer se putem te hijerarhije potom mogu rješavaju mnogi problemi koji dolaze u procesu izrade video igara. Elementima je često potreban nadzor i upravljanje nad drugim elementima. Takvi elementi su zatim u pravilu roditelji.

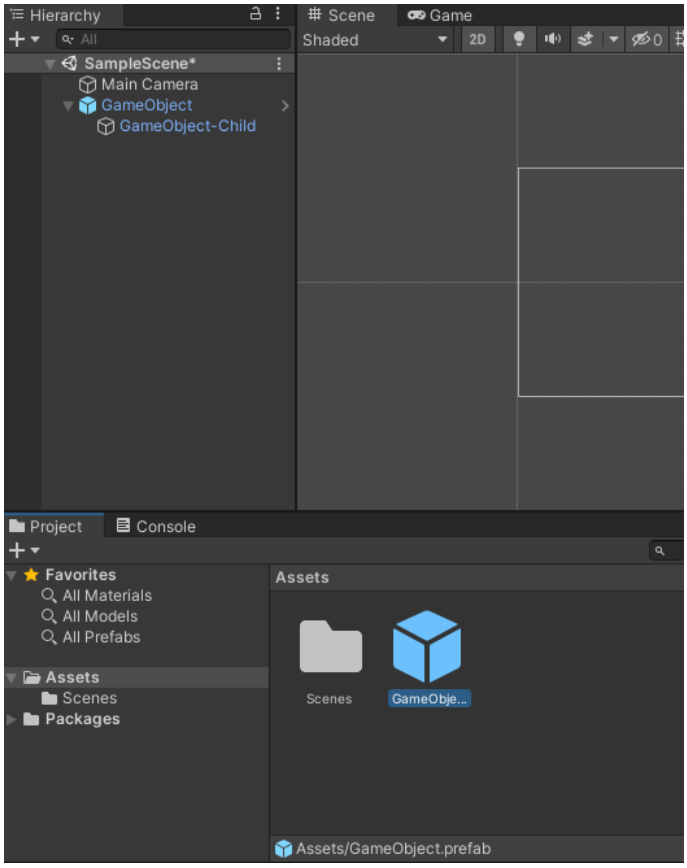
Elementi kakve želimo finalno postići se jako često moraju podijeliti na manje elemente te biti grupirati. Tako se može jednostavnije manipulirati njima te ih mijenjati nego li da se radi o jednom objektu. Takvi elementi su u pravilu djeca nekom roditelju. Kreiramo li novi objekt igre unutar već postojećega, kreirati ćemo odnos roditelj-dijete.



Slika 6: Prikaz odnosa roditelj-dijete unutar hijerarhijskog preglednika

3.6. Spremanje objekata za ponovno korištenje

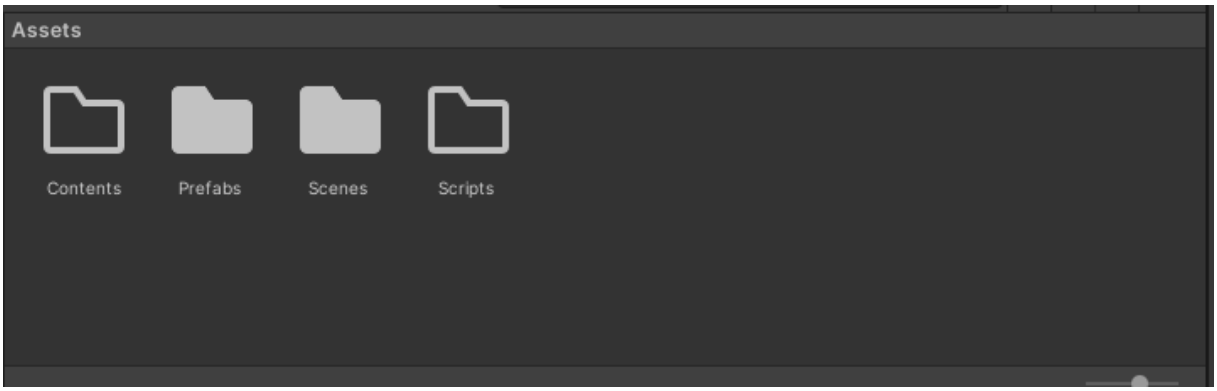
Objekt se sa hijerarhijskog stabla može spremiti za druge potrebe i situacije. Dovoljno je povući mišem objekt sa hijerarhijskog preglednika u projektni preglednik, na željeno mjesto. Primijetimo da je objekt koji je prebačen u projektni direktorij tako poplavio na hijerarhijskom pregledniku. Kao takav, siguran je za brisanje sa preglednika.



Slika 7: Prikaz objekta spremljen za ponovno korištenje

Preporučljivo je i organizirati temeljni direktorij tako da ga se podijeli na pod-direktorije gdje ćemo spremati svoje projektne datoteke. Mi smo na početku u praktičnom dijelu podijelili direktorij na *Scenes, Scripts, Prefabs i Contents* direktorije.

Možemo zaključiti da u *Scenes* direktorij postavljamo sve scene, a u *Scripts* skripte. U *Prefab* direktorij ćemo spremati sve objekte igre, a u *Contents* ćemo staviti sve grafičke resurse koji nam budu potrebni za kreiranje grafičkog sadržaja.



Slika 8: Prikaz projektnog direktorija

3.7. Skriptiranje

„Skriptiranje je bitan sastojak u svim aplikacijama koje napravite u Unityju. Za većinu aplikacija potrebne su skripte.“ Uloge skripti su „odgovoriti na ulaz igrača i dogovoriti da se događaji u igri odigraju kad bi trebali. Osim toga, skripte se mogu koristiti za stvaranje grafičkih efekata, kontrolu fizičkog ponašanja objekata ili čak implementaciju prilagođenog AI sustava za likove u igri.“ [5].

3.7.1. Skripta kao komponenta – MonoBehaviour

Svi objekti igre se mogu proširivati sa skriptama. Tada se skripte ponašaju kao kompontente unutar objekta. Da bi skripta postala komponenta objektu, potrebno ju je proširiti s MonoBehaviour klasom. Naime, unutar ove klase se nalaze sve druge skripte koje Unity implementira. Nasljeđivanjem atributa i metoda, skripta poprima pristup metodama Start i Update koje su esencija svih komponenata objekta. Start metoda se izvršava jednom, kada je objekt instanciran. Update metoda se za svaku sličicu koja se izvrti u sekundi. Drugim riječima, ako se igra vrti na 60 sličica u sekundi, tada se Update metoda okine 60 puta. Pored Start i Update metode postoji i Awake metoda. Ona se okine svaki put kada se objekt promjeni

iz neaktivnog u aktivno stanje. Objekti imaju i svoj životni vijek. Svaki put kada je objekt uništen, poziva se OnDestroy metoda.

```
using UnityEngine;

Unity Script | 0 references
public class GameObjectScript : MonoBehaviour
{
    Unity Message | 0 references
    void Start()
    {
        Debug.Log("Prije prve obnove okvira");
    }

    Unity Message | 0 references
    void Update()
    {
        Debug.Log("Okvir");
    }

    Unity Message | 0 references
    private void Awake()
    {
        Debug.Log("Probudio sam se!");
    }

    Unity Message | 0 references
    private void OnDestroy()
    {
        Debug.Log("Umirem!");
    }
}
```

Slika 9: Isječak programskog koda koji prikazuje Start, Update, Awake te OnDestroy metode

3.7.2. Dohvačanje drugih komponenata

Kako skripta ujedno predstavlja i komponentu objekta, tako ona ima pristup i ostalim komponentama objekta. Na primjer, jedna skripta može pozvati drugu. Dovoljno je upotrijebiti *this* objekt koji predstavlja objekt igre. Jedna od važnih metoda koje objekti igre sadrže je *getComponent()*. Pomoću nje možemo pozvati drugu skriptu na sljedeći način:

```
var objektDrugeKlase = this.getComponent<ImeDrugeKlase>();
```

Prema tome, ako u drugoj klasi imamo definiranu metodu *DajDva* koja vraća vrijednost 2:

```
public int DajDva(){ return 2; }
```

Unutar prve skripte metodi (i njenoj povratnoj vrijednosti) možemo pristupiti sa sljedećom linijom koda:

```
int dva = this.getComponent<ImeDrugeKlase>().DajDva();
```

Pomoću ovog jednostavnog primjera možemo zaključiti kako se skriptiranjem mogu dohvaćati i ostale vrijednosti svih komponenata unutar objekta. Na primjer Transform komponenta:

```
var transform = this.transform;
```

Skripte kao komponente roditeljskog objekta su u mogučnosti dohvaćati i skripte kao komponente objekata djece tog roditelja.

```
Transform transform = this.transform.Find(„GameObject-Child“);
```

Radi bolje organizacije projektnog važno je da objekti imaju hijerarhijsku organizaciju koja će omogućiti da roditelj manipulira djecom. Posebice kod dinamičkog instanciranja objekata.

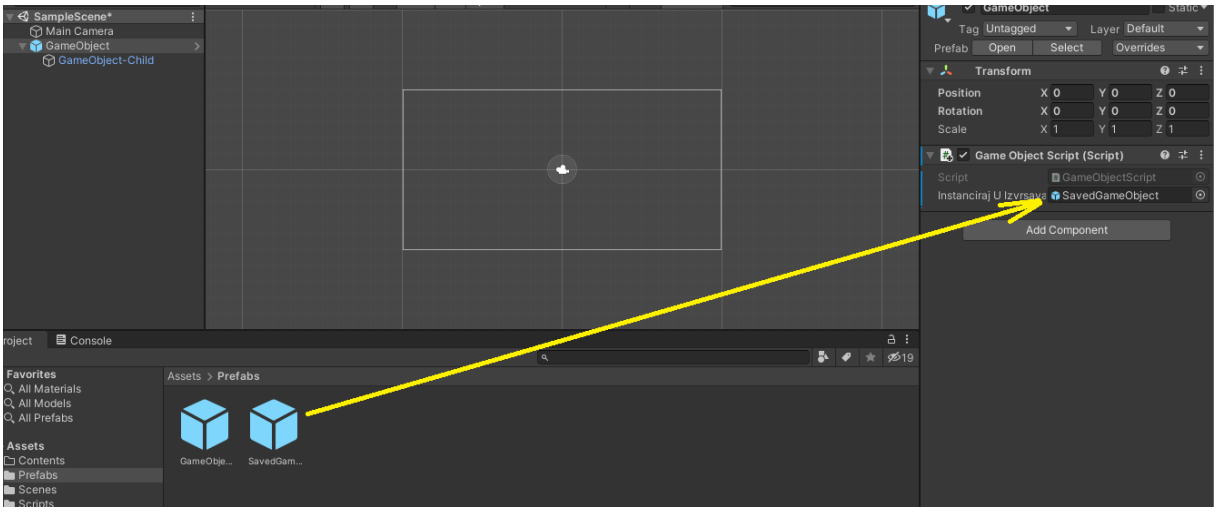
3.7.3. Dinamičko instanciranje objekata

Implementacija video igre podrazumijeva dinamičko instanciranje objekata. Ponekad je dobro upravljanje životom objekata krucijalno za performanse igri. Moderne igre su itekako zahtjevne te tehnologija jako brzo napreduje u snazi. Sa dinamičkim instanciranjem objekata moguće je kreirati objekt netom prije nego nam on zatreba. Unutar skripte kao komponente objekta moguće je u izvođenju instancirati prazni objekt igre:

```
GameObject gameObject = new GameObject();
Instantiate(gameObject, Vector3.zero, Quaternion.identity);
```

Prazan objekt nam u izvođenju nam baš i nije koristan, no možemo kreirati objekt koji smo prethodno spremili i referencirali na skriptu.

Da bi uspješno referencirali spremljeni objekt, potrebno je unutar skripte roditelja kreirati javni objekt *GameObject* klase. Taj objekt će se prikazati u pregledniku komponenata pa zatim treba iz projektnog direktorija dovući spremljeni objekt koji želimo instancirati.



Slika 10: Referenciranje spremljenog objekta na skriptu

Nakon uspješnog referenciranja objekta potrebno je unutar skripte pozvati metodu *Instantiate*.

3.7.4. Prikaz više klonova objekata korisničkog iskustva

Dinamičko instaniranje objekata je jednostavno, no postavlja se pitanje kako ćemo ih prikazati na ekranu. Dinamički instancirani objekt je moguće programski postaviti kao dijete elementa. To radimo sa metodom *SetParent* iz klase *Transform.*

```
kloniraniObjekt.transform.SetParent(roditeljObjekta.transform);
```

Pogledajmo kako to izgleda u praksi. Prvo smo kreirali *Panel* objekt korisničkog iskustva. Postavili smo ga kao dijete *Canvas* objekta. Finalno smo referencirali *Canvas* sa skriptom *GameObjectScript* koja je komponenta roditelja svih elemenata.

Također smo sa skriptom referencirali i spremljeni objekt igre *SavedGameObject.* Finalno smo u *Update* metodi skripte pozvali *Instantiate* metodu za instanciranje objekata te *SetParent* metodu za postavljanje objekta kao dijete drugom elementu, u našem slučaju *Panel* objektu.

```
Unity Message | 0 references
void Update()
{
    GameObject klon = Instantiate(instancirajUIzvrsavanju, Vector3.zero, Quaternion.identity);
    klon.transform.SetParent(panel.transform);

}
```

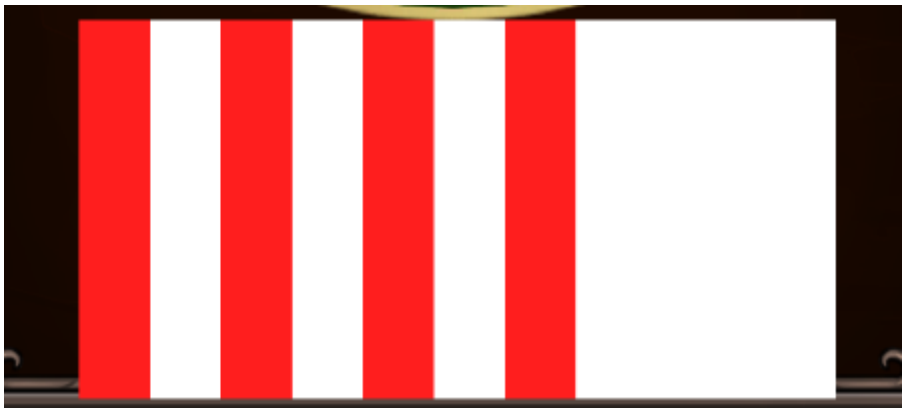
Slika 11: Isječak koda za instanciranje objekata u izvođenju

Primječujemo da je i postavljanje roditelja instanciranom objektu lagan zadatak, no postavlja se pitanje kako ćemo prikazati više instanciranih objekata u ispravnom položaju, kako bi korisničko iskustvo bilo na najvišoj razini. Pozicioniranje takvih elemenata u scenu može biti problem, no za to nam služe komponente *Horizontal layout group* te *Vertical layout group*.

3.7.4.1. Horizontal layout group

Za prikaz elemenata u horizontalnom redosljedu brine se Horizontal layout group komponenta. Da bi se objekti kao djeca roditelja instancirali sa ispravnim pozicioniranjem, dovoljno je na objekt roditelja pridružiti ovu komponentu.

Dodajmo u Panel objekt, koji je dijete dinamički instanciranih objekata ovu komponentu. Komponenta se pridružuje objektom klikom na *Add Component* gumb unutar preglednika komponenata, te pretragom ove komponente. Također je potrebno namjestiti vrijednosti ove komponente po našem izboru. Mi ćemo postaviti *Spacing* vrijednost da bude jednaka -200. *Control Child Size* i *Child Force Expand* ćemo označiti za *Width* i *Height*.



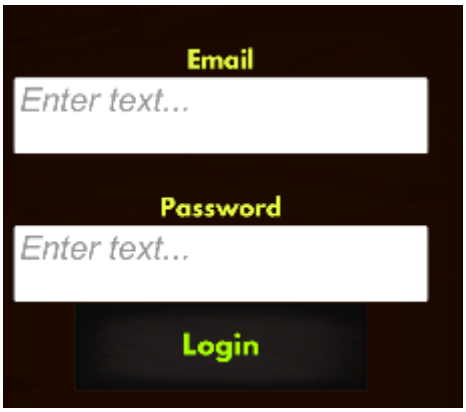
Slika 12: Prikaz pozicioniranih objekata u Horizontal layout group

Pridruživanjem ove komponente roditeljskom elementu možemo primjetiti da će se djeca elementa instancirati jedan pored drugoga, u horizontalnom redosljedu.

3.7.4.2. Vertical layout group

Slično kao i za *Horizontal layout group*, ova komponenta se brine sa ispravno pozicioniranje elemenata djece, ali u vertikalnom obliku. Ova komponenta je jako korisna kada želimo grupirati elemente jedan ispod drugoga

Tako smo na primjer, u praktičnom dijelu kreirali formu za prijavu. Svaki element koji je dijete roditelja se pozicionira u vertikalnom redosljedu jednom ispod drugoga. Izmjenjuju se tekst, polja za unos te gumb.



Slika 13: Prikaz pozicioniranih elemenata u vertical layout group

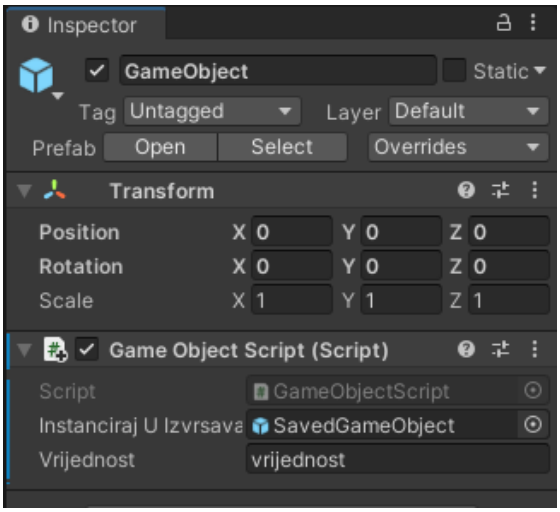
3.7.5. Serijalizacija vrijednosti

Serijalizacija je iznimno korisna, posebice za testiranje. Unutar vlastitih skripti moguće je serijalizirati vrijednost koja će potom biti prikazana unutar preglednika komponenata. Dovoljno je iznad atributa dodati SerializeField atribut:

```
[SerializeField]
public string vrijednost = "vrijednost";
```

Slika 14: Isječak koda koji prikazuje kako se vrši serijalizacija vrijednosti

Sada ovoj vrijednosti možemo pristupiti i unutar Unity sučelja pregledom objekta na pregledniku. Također, vrijednost je moguće i mijenjati u izvođenju, što nam govori zašto je korisno za testiranje.



Slika 15: Prikaz serijalizirane vrijednosti unutar Unity programskog alata

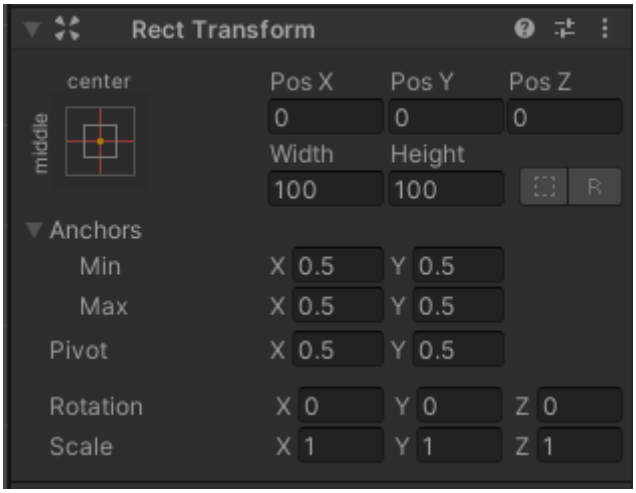
3.8. Objekti korisničkog sučelja

Objekti korisničkog sučelja (eng. *user interface - UI*), su već predifinirani objekti koje možemo iskoristiti za kreiranje korisničkog sučelja. Među takve objekte spada Gumb (eng. *Button*), Slika (eng. *Image*), Panel, Kanvas (eng. *Canvas*), Text (eng. *Tekst*), i ostali. Drugim riječima, postoje već kreirane kompontente koje su im pridružene. Za kreiranje UI objekta dovoljno je desnim klikom na hijerarhijskom pregledniku proširiti UI i odabrati željeni element.

3.8.1. RectTransform

Zajednička značajka UI objekata je ta da dijele komponentu *RectTansform*, koja proširuje Transform metodu objekata igre. Za razliku od Transform komponente koju sadržava primjerice prazan objekt, *RectTransform* ima dodatne vrijednosti i zanimljive osobine. „Komponenta Rect Transform je 2D izgled pandana komponenteTransform. Tamo gdje Transform predstavlja jednu točku, Rect Transform predstavlja pravokutnik u koji se može staviti element korisničkog sučelja. Ako je roditelj Rect transformacije također Rect Transform, podređena Rect Transform može također odrediti kako bi trebala biti postavljena i veličine u odnosu na roditeljski pravokutnik.“ [6].

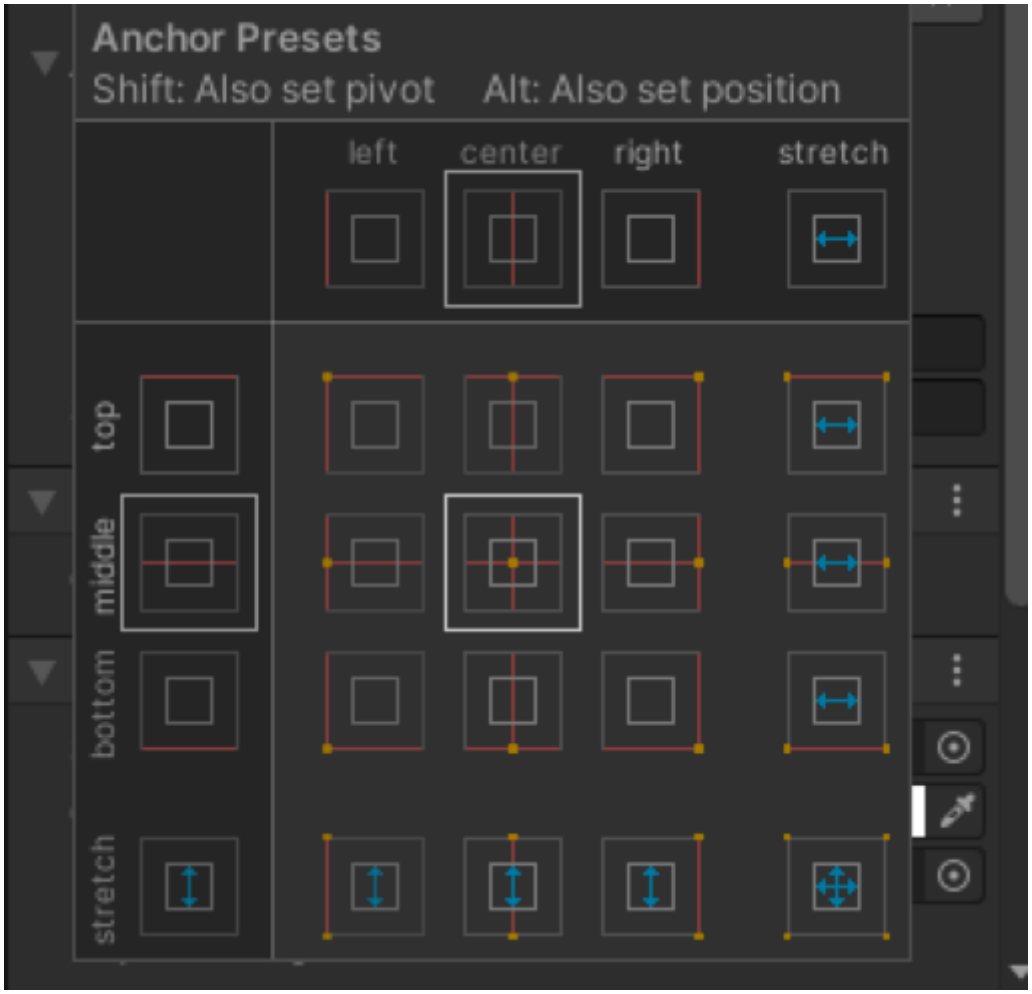
Ova komponenta je izrazito korisna za rješavanje problema responzivnosti elemenata unutar igre. Različiti uređaji zbog svojih dimenzija mogu prikazati elemente igre na različit način, a *RectTransform* se može pobrinuti da se na različitim dimenzijama objekti korisničkog sučelja prikažu na ispravan način.



Slika 16: Prikaz RectTransform komponente na hijerarhijskom pregledniku

Unutar sučelja možemo primijetiti da se može mijenjati položaj objekta u odnosu na roditelja. Ovisno o položaju koji odaberemo, referentna točka položaja će se promijeniti.

Mijenjanje položaja je korisno ovisno o svrsi. Primjerice, kada želimo prikazati listu UI objekata, korisno je da su u centru roditelja.



Slika 17: Način promjene položaja objekta djeteta u odnosu na roditelja

3.8.2.Canvas

Canvas je izrazito važan element koji djeluje kao omotač svim drugim UI objektima. Dakle, unutar hijerarhijskog stabla, Canvas objekt je roditelj svim drugim UI elementima. Također, kreiranjem Canvasa, Unity automatski kreira EventSystem objekt koji osluškuje događaje UI objekata.

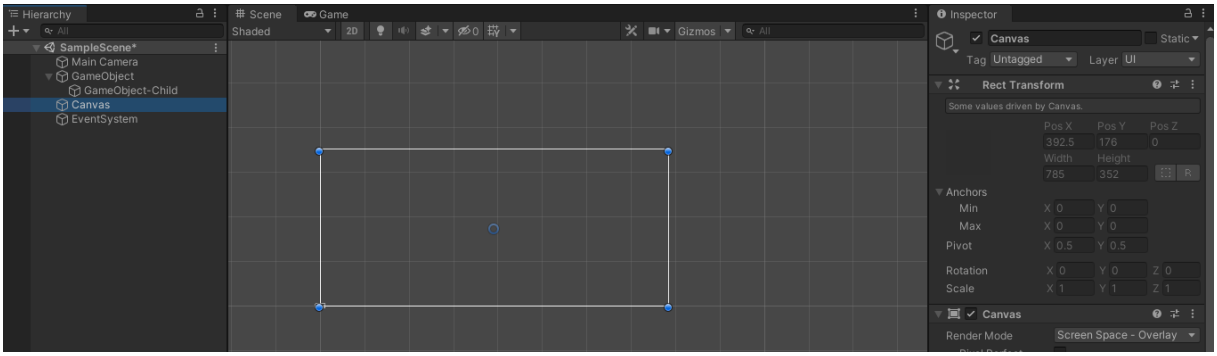
Canvas objekata može biti više u hijerarhijskom pregledniku te svaki canvas može odrediti različit način renderanja UI objekata djece. Postoje tri različita načina:

- Screen space – Overlay
- Screen space – Camera
- World space

Prvi oblik, odnosno *Screen space – overlay* je uobičajeni oblik prikaza koji je automatski postavljen od strane Unitya. Kod ovog oblika prikaza, elementi se pozicioniraju i prikazuju ovisno o rezoluciji ekrana. Dakle, pozicioniraju se prema dimenzijama ekrana. Drugim riječima, koordinate elementa kojeg se pozicionira se translatiraju prema dimenzijama ekrana.

Screen space – Camera se za razliku od Overlay-a brine za to da se UI elementi unutar Canvasa prikazuju prema glavnoj kameri koja se nalazi na sceni. Drugim riječima, koordinate elementa kojeg se pozicionira se translatiraju prema kameri koja obuhvaća svijet unutar scene.

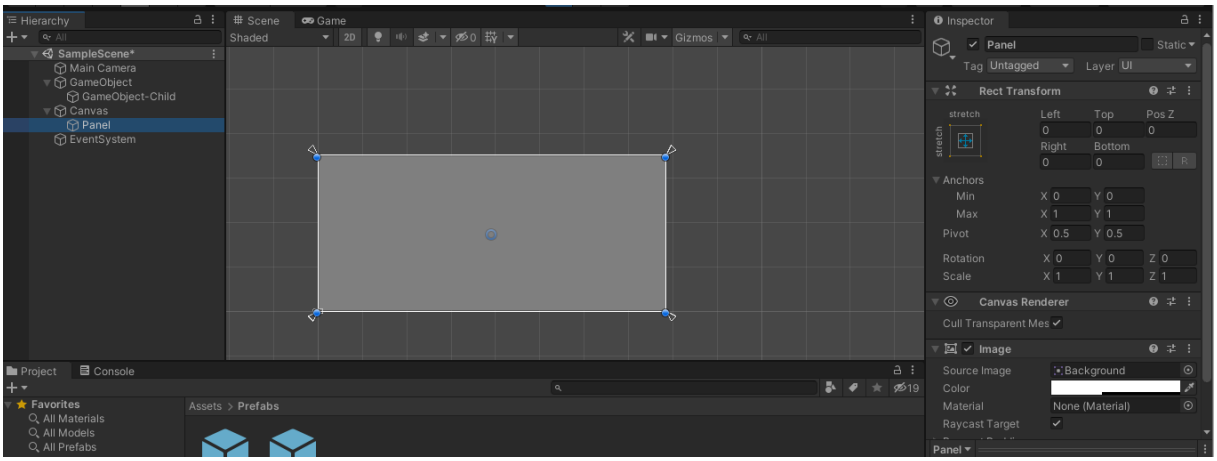
World space je oblik kod kojega se ne mari na dimenzije ekrana ili veličinu prostora koji obuhvaća kamera. Kod ovog oblika se gledaju dimenzije cijelog prostora koji postoji unutar scene. Kod igara visoke interaktivnosti sa velikim mapama često kamera i dimenzije ekrana nisu dovoljno velike za sve što se može prikazati. Sa ovim oblikom možemo pozicionirati UI elemente negdje unutar tog svijeta. Finalno, kod ovog oblika se koordinate elementa translatiraju prema svijetu koji se nalazi u sceni.



Slika 18: Prikaz Canvas objekta unutar scene

3.8.3.Panel

Panel se preporučuje koristiti kada želimo grupirati više elemenata unutar jednog. Panel je potom najbolje koristiti kao roditelj svim UI elementima koji se povezuju u jednu cjelinu. To radi na način da djeca ovog elementa translatiraju svoje koordinate na centar ovog elementa. S druge strane, Panel translatira sebe uz rubove Canvas elementa koji mu je roditelj. Element dolazi sa predkonfiguriranim objektom slike (Image) koja može predstavljati pozadinu prostora unutar kojega su UI elementi grupirani.



Slika 19: Prikaz Panel objekta unutar scene

3.8.4. Slika

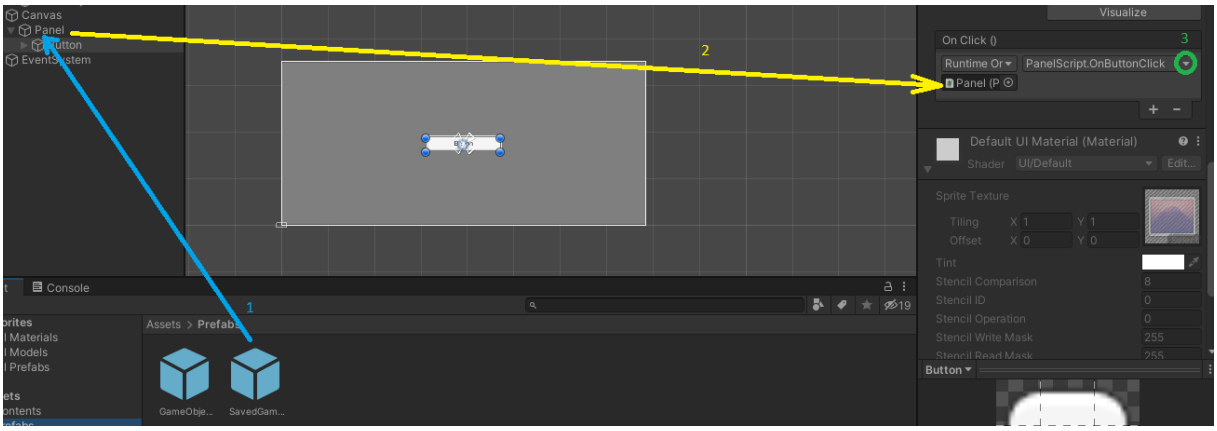
Slično Panelu, slika je objekt koji ima na sebi Image komponentu, odnosno komponentu slike. Ipak, kao element, nije zamišljen da grupira druge elemente zbog toga što se pozicionira u centar svoga roditelja.

3.8.5. Gumb

Gumb je klasični UI element koji na događaj klika izvršava neku akciju. Kako smo već spomenuli, EventSystem objekt je zadužen za slušanje svih događaja. To znači da ako se gumb ne nalazi unutar Canvas elementa, on jednostavno ne bi radio. Da bi povezali klik na gumb sa nekom operacijom, moramo se pobrinuti za nekoliko stvari.

1. Kreirati metodu unutar neke skripte.
2. Skriptu, kao komponentu pridružiti jednom od roditeljskih objekata.
3. Objekt kojemu je pridružena skripta referencirati na OnClick komponentu i odabrati željenu metodu

Unutar Unity programa referenciranje se može izvršiti povlačenjem roditeljskog objekta sa hijerarhijskog prozora u onClick metodu unutar komponente gumba. Ukoliko smo uspješno izvršili referenciranje, metoda koju smo kreirali unutar roditeljskog objekta će se pozvati nakon klika na gumb.



Slika 20: Referenciranje metode za okidanje događaja na klik gumba

Gumb se može konfigurirati i tokom izvršavanja aplikacije. Naime, postoje situacije kada je potrebno gumb dinamički instancirati. Tada je potrebno onClick metodu konfigurirati programski.

OnClick je po svojoj prirodi samo slušač koji poziva određeni blok programa nakon što se dogodi događaj klika na gumb. Programeri u taj blok programa mogu upisati svoju skriptu te će se ista izvršiti nakon događaja.

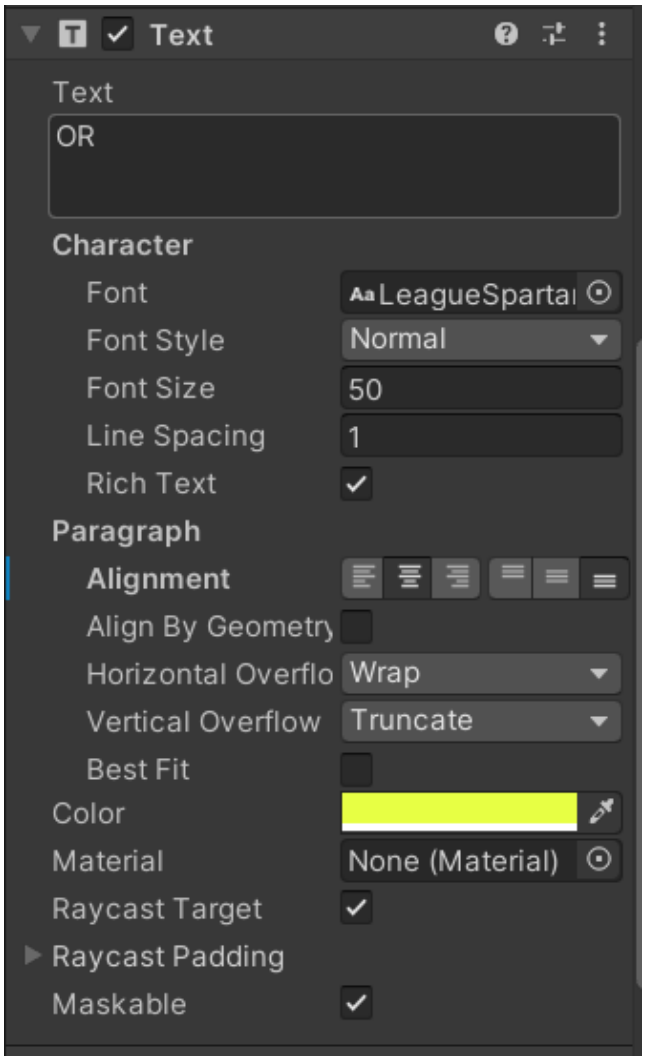
U praktičnom dijelu rada imali smo situaciju gdje smo dinamički morali kreirati klonove Panela sa svojom djecom. Svaki panel ima svoj gumb. Dakle, morali smo programski narediti gumbu što da izvrši nakon događaja klika. Pogledajmo dinamičko kreiranju slušača izgleda.

```
This.getComponent<Button>().onClick.AddListener(() => {});
```

3.8.6. Tekst

Ovaj element očekivano, prikazuje tekst. Engleski se naziva Text te je jako važan za korisničko iskustvo. Kada kreiramo gumb, tekst element dolazi pred-definirano sa gumbom kao njegovo dijete. Na taj način gumb poprimi tekst na sebi.

Ipak, ovaj element se može i samostalno kreirati. Ako obratimo pažnju, unutar preglednika komponenata možemo pronaći Text komponentu pomoću koje manipuliramo cijelim izgledom teksta. Ovdje možemo promijeniti vrijednost teksta, boju, oblik, strukturu, veličinu, način prikaza u odnosu na veličinu kontenjera za pisanje, i ostalo.



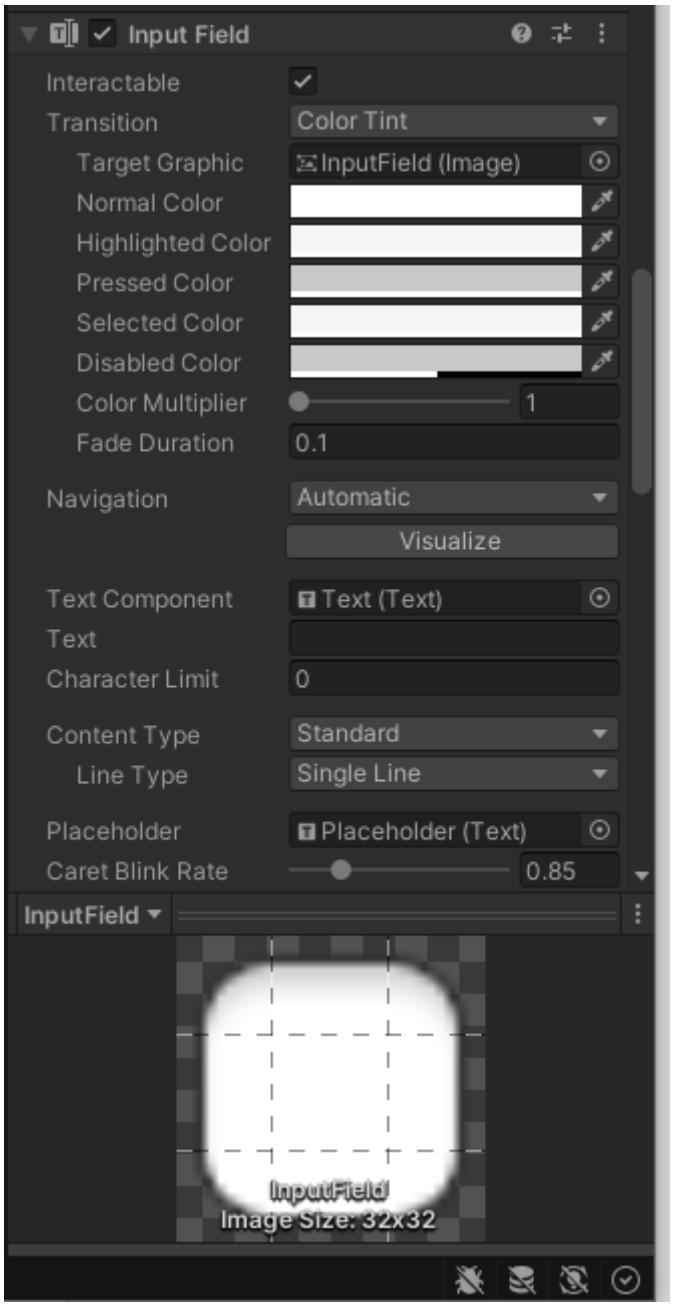
Slika 21: Prikaz Text komponente unutar preglednika komponenata

3.8.7. Polje za unos

Na engleskom jeziku je poznatiji kao Input Field. Ovaj element je važan kada trebamo od korisnika da unese neku vrijednost, najčešće putem tipkovnice. Dakle, zaključujemo da se ovaj element koristi kod prijave i registracije. Potreban nam je za unos korisničkog imena, lozinke te ostalih informacija.

Cijeli UI element sadržava na sebi Input Field komponentu gdje možemo odrediti boju i izgled cijelog elementa, kao i tekst. Također, putem komponente možemo upravljati preddefiniranim događajima.

Ovaj element ima i dvoje djece – *placeholder* i *text*. Oba elementa su zapravo tekst elementi te je jednome zadatak prikazati pred-definirani tekst na elementu („Unesite lozinku“), a drugome je zadatak prikazati tekst koji se unosi u realnom vremenu.



Slika 22: Prikaz Input Field komponente unutar preglednika komponenata

4. Implementacija potrebnih funkcionalnosti

U ovom poglavlju ćemo proći kroz implementaciju bitnih dijelova igre za više igrača. Među njih spadaju sama prijava te registracija računa. Ispravna autentifikacija te autorizacija, povezivanje igrača u predvorje, sobu te finalno u igru. Zvuči jednostavno ali ovdje postoji jako puno procesa koji poboljšavaju korisničko iskustvo te prema tome moraju biti implementirani. Na primjer, izbacivanje igrača iz sobe, informiranje o spremnosti, komunikacija između igrača, i ostalo.

4.1. Upravljanje korisničkih računom

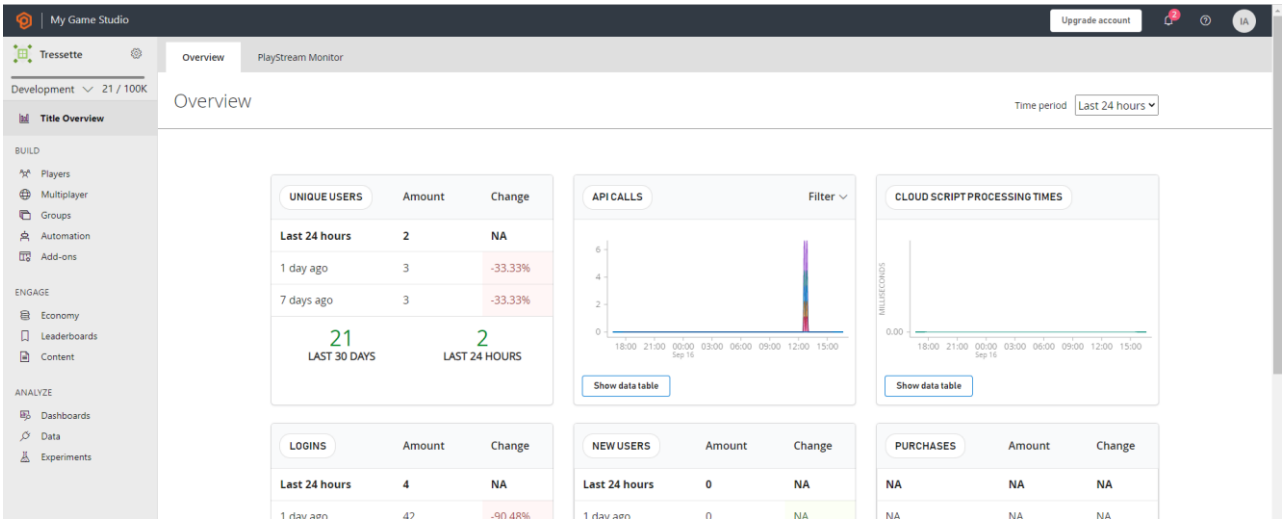
S obzirom da razvijamo igru za više igrača, moramo se pobrinuti da su njihovi korisnički računi sigurni te da su svi procesi od njihove authentifikacije do igranja igre što jednostavniji.

4.1.1. Playfab

Kako za naš projekt nije bilo važno spremanje korisničkih podataka u bazu, pobrinuli smo se za implementaciju procesa autentifikacije i autorizacije korištenjem Playfab API-a. „PlayFab je potpuna pozadinska platforma za igre uživo s upravljanim uslugama igara, analitikom u stvarnom vremenu..“ [7]

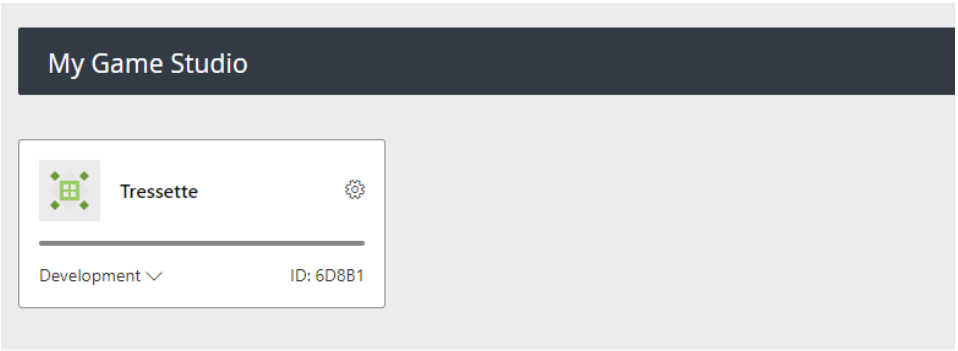
Ukratko, Playfab je cloud sustav razvijen od strane Microsoft-a, koji ima puno riješenih problema, povezanih uz razvoj video igara za više igrača. Sa besplatnom verzijom sustavu imamo pristup Playfab API-u koji nam pruža neke značajke poput registracije korisnika te njihov login, kao i kreiranje novih predefiniranih vrijednosti o korisnicima.

Proces kreiranja računa za Playfab je jednostavan. Nakon prolaska kroz čarobnjak, korisnik će biti registriran te će mu biti ponuđena opcija kreiranja novog projekta. Kreiranjem projekta otvara nam se cijeli preglednik tog projekta.



Slika 23: Playfab korisničko web sučelje

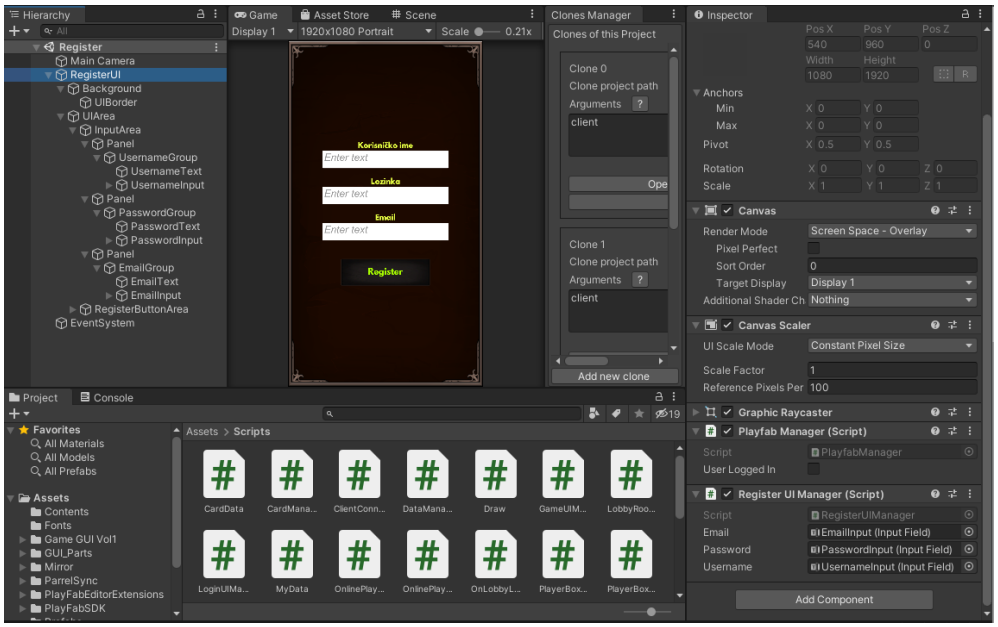
Također, pojavio nam se naš studio koji smo kreirali unutar općeg preglednika svih projekata. ID koji je zapisan je važan za naš razvoj jer ćemo se putem njega spajati na Playfab API.



Slika 24: Prikaz studija rezerviranog za video igru koju razvijamo

4.1.2. Login i registracija

Prvo smo razvili grafičko sučelje za Registraciju. Osnovnu strukturu dobili smo korištenjem tri Panel objekata unutar jednog roditeljskog objekta koji sadrži *Vertical Layout Group* komponentu. Unutar panela postavili smo elemente korisničkog iskustva tekst i polja za unos kako bi korisnik unio



Slika 25: Prikaz forme za registraciju unutar Unity programskog alata

Iz preglednika komponenata možemo primijetiti da RegisterUI objekt sadrži skripte *PlayfabManager i RegisterUIManager* kao komponente. *RegisterUIManager* je klasa koja ima referencirane objekte unosa te metodu koja osluškuje klik na gumb „Register“.

```
public class RegisterUIManager : MonoBehaviour
{
    public InputField Email;
    public InputField Password;
    public InputField Username;
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {

    }

    0 references
    public void OnRegisterButtonClick()
    {
        Debug.Log("register klik!");
        this.GetComponent<PlayfabManager>().Register(this.Email.text, this.Username.text, this.Password.text);
    }
}
```

Slika 26: RegisterUIManager skripta te metoda OnRegisterButtonClick koja osluškuje gumb za registraciju

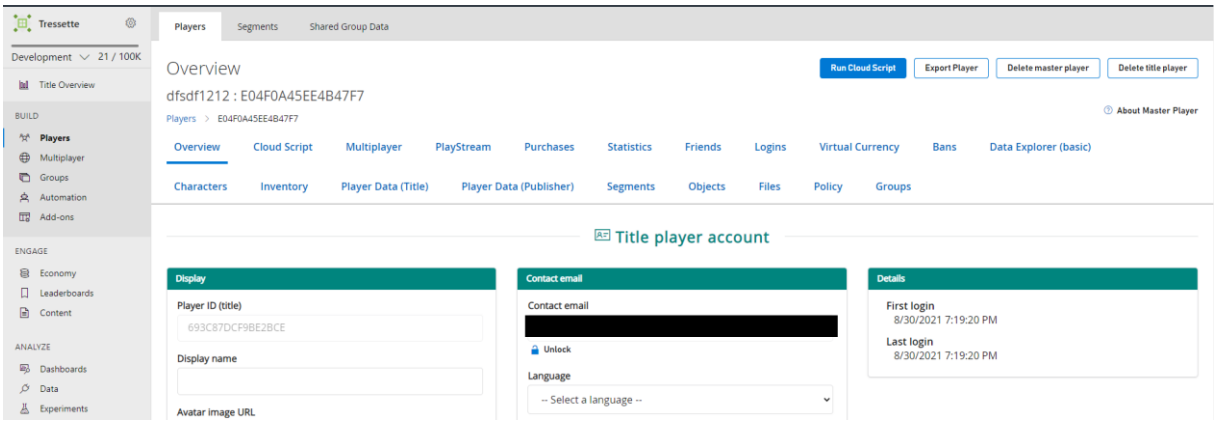
Klasu *PlayfabManager* namijenili smo izvršavanju prijave i registracije putem Playfab API-a. Za registraciju je zaslužna metoda *Register.*

```
1 reference
public void Register(string email, string username, string password)
{
    PlayFabSettings.TitleId = "6D8B1";
    this.Email = email;
    this.Username = username;
    this.Password = password;
    var register = new RegisterPlayFabUserRequest { Email = this.Email, Password = this.Password, Username = this.Username };
    PlayFabClientAPI.RegisterPlayFabUser(register, OnRegisterSuccess, OnRegisterFailure);
}
```

Slika 27: Register metoda za registraciju korisnika

Primjetimo da metoda *RegisterPlayfabUser*, koja dolazi od Playfab API-a, sadržava dvije metode *OnRegisterSuccess i OnRegisterFailure,* koje se izvršavaju ovisno o tome je li je registracija u oblaku uspjela, ili ne. Izvođenjem

S uspješnom registracijom *RegisterPlayfabUser* metode, kreira se novi račun unutar Playfab administrativnog sučelja.



Slika 28: Prikaz kreiranog igrača unutar Playfab web sučelja

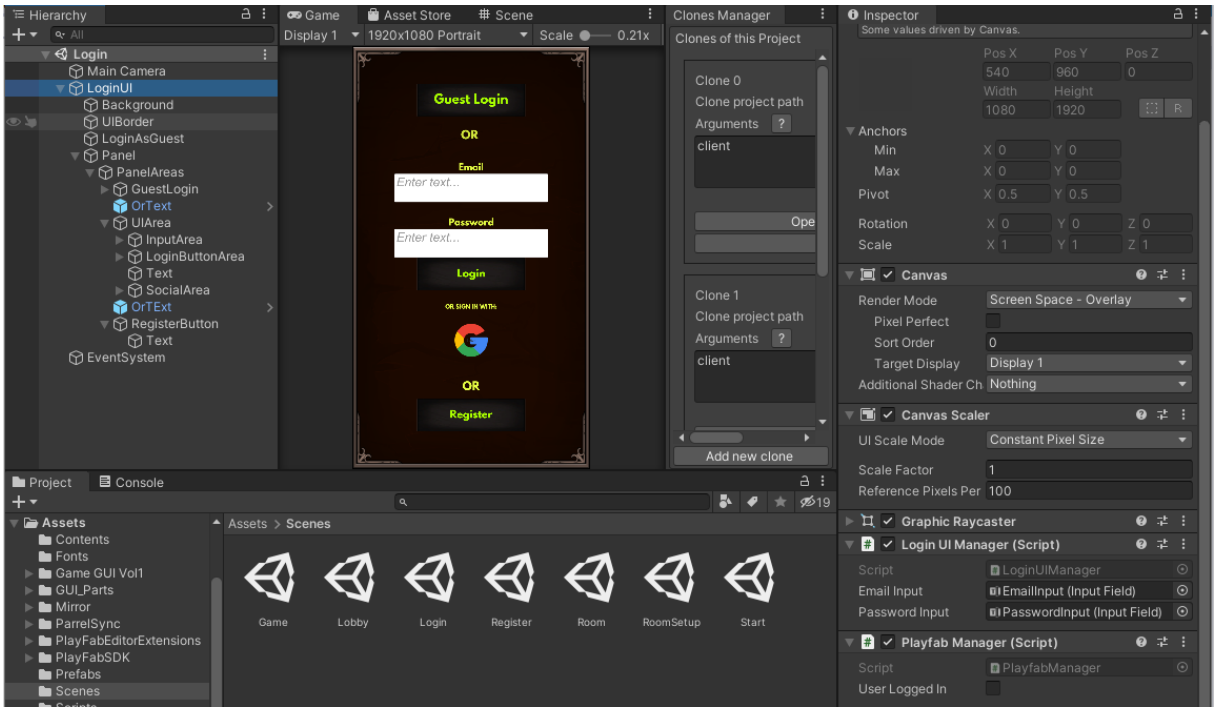
U *OnRegisterSuccess* primili smo objekt kao odgovor uspješnog zahtjeva za registracijom. Zahtjev se poslao putem HTTPS protokola, a odgovor se nalazi u *RegisterPlayfabUserResult* objektu.

```
1 reference
private void OnRegisterSuccess(RegisterPlayFabUserResult obj)
{
    TcpKlijent klijent = new TcpKlijent();
    DataManager dm = new DataManager();
    dm.SetMySessionTicket(obj.SessionTicket);
    dm.SetMyPlayfabId(obj.PlayFabId);
    klijent.PosaljiServeru("{\"commandId\":\"YOIJUSTREGISTERED\", \"sessionTicket\":\"" + obj.SessionTicket + "\"}");
    string odgovor = klijent.PrimiOdServera();
    dm.SetPlayerLoggedIn(true);
    SceneManager.LoadScene("Lobby");
}
```

Slika 29: Metoda OnRegisterSuccess

Unutar objekta *obj*, možemo pristupiti vrijednosti *obj.SessionTicket*, koja sadrži hashiranu vrijednost koja će nam poslužiti za autorizaciju. Na samom kraju uspješne registracije, mijenjamo scenu u *Lobby*, odnosno scenu za predvorje.

Sljedeću scenu koju smo napravili je scena za login. Ideja je da finalni proizvod sadrži prijavu kao gost, putem vlastitog email računa, ili putem Google autentifikacije.



Slika 30: Forma za prijavu unutar programskog alata Unity

Još jednom kombinacijom *Panel* objekta igre i *Vertical Layout Group komponente* uspjeli smo dobiti grafičko sučelje koje se dijeli na nekoliko dijelova. Jednu grupu čini gumb za prijavu kao gost te posebnu grupu čine tekstovi „OR“. Nadalje, jednu grupu čine grupirani elementi za unos email-a te lozinke, sa svojim pripadajućim tekst objektima i gumbom *Login* Finalno, i prijava sa Google računom čini posebnu grupaciju objekata korisničkog iskustva.

Obratimo pažnju na LoginUI objekt. Primjećujemo da i on ima pridruženu skriptu *PlayfabManager* na sebi, kao i *RegisterUIManager* objekt u sceni za registraciju. To je zato što će se finalno pozvati *Login* metoda klase *PlayfabManager.*

```
1 reference
public void Login(string email, string password)
{
    PlayFabSettings.TitleId = "6D8B1";
    this.Email = email;
    this.Password = password;
    Debug.Log(email);
    Debug.Log(password);
    var login = new LoginWithEmailAddressRequest { Email=this.Email, Password=this.Password };
    PlayFabClientAPI.LoginWithEmailAddress(login, OnSuccess, OnError);
}
```

Slika 31: Metoda Login koja prijavljuje korisnika u sustav

Ovdje se korištenjem HTTPS protokola poziva REST servis za prijavu na Playfab. Korisnici unosom svoga email-a i lozinke šalju zahtjev za prijavom u igru. Ukoliko je korisnik unio ispravne podatke, izvršiti će se OnSuccess metoda.

```
1 reference
private void OnSuccess(LoginResult result)
{
    Debug.Log(JsonUtility.ToJson(result));
    Debug.Log("Successfully logged in");
    this.GetLeaderboard();
    this.UserLoggedIn = true;
    GetUserData(result.PlayFabId);
    TcpKlijent klijent = new TcpKlijent();
    DataManager dm = new DataManager();
    dm.SetMySessionTicket(result.SessionTicket);
    dm.SetMyPlayfabId(result.PlayFabId);
    klijent.PosaljiServeru("{\"commandId\":\"YOIJUSTLOGGEDIN\", \"sessionTicket\":\"" + result.SessionTicket+"\"}");
    string odgovor = klijent.PrimiOdServera();
    LoginResponse parsedResponse = JsonConvert.DeserializeObject<LoginResponse>(odgovor);
    Debug.Log("Primio sam odgovor:" + odgovor);
    klijent.ZatvoriSocket();
    dm.SetMyUsername(parsedResponse.Username);
    dm.SetPlayerLoggedIn(true);
}
```

Slika 32: OnSuccess metoda

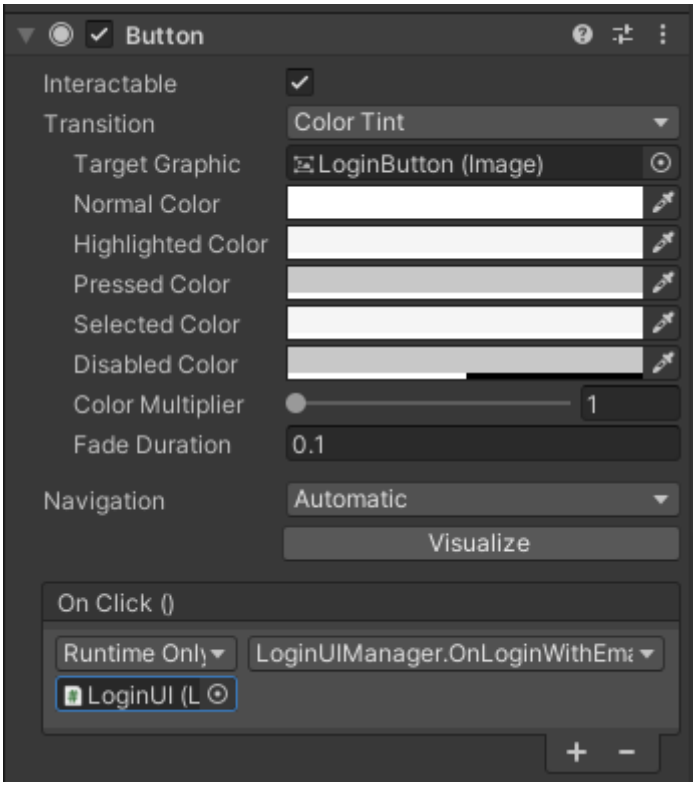
Nakon uspješne prijave, sa *LoginResult* objektom možemo obraditi podatke iz odgovora od servisa. I ovdje dobijamo SessionTicket vrijednost unutar odgovora koju potom šaljemo pozadinskom servisu na autorizaciju i obradu daljnjih procesa.

Da bi igra znala da korisnik zahtjeva prijavu, objektu LoginUI je pridružena i *LoginUIManager* skripta koja na identičan način kao i kod registracije osluškuje gumb za prijavu te ima referencirane objekte za unos.

```
0 references
public void OnLoginWithEmailButtonClick()
{
    this.GetComponent<PlayfabManager>().Login(EmailInput.text, PasswordInput.text);
}
```

Slika 33: Metoda OnLoginWithEmailButtonClick koja osluškuje gumb za prijavu

Metodu *OnLoginWithEMailButtonClick()* smo referencirali na *OnClick* događaj tako što smo iz hijerarhijskog preglednika mišem povukli *GameUI* objekt sve do *Button* komponente gumba, koji je dijete *LoginUI* objekta.



Slika 34: Prikaz Button komponente za prijavu

Na ovaj način smo osigurali da će se uslijed klika na gumb svaki put pozvati *OnLoginWithEmailButtonClick* metoda roditelja ovog gumba.

4.1.3. Autorizacija

Spomenuli smo da se kod uspješne prijave i registracije korisniku vraća njegov pripadni objekt. Primjetili smo da se iz pripadnih objekata može pročitati *SessionTicket* vrijednost tipa podatka string.

Ovo polje znakova predstavlja hashirani token s kojim možemo autorizirati korisnika za korištenje određenih resursa našeg pozadinskog servisa.

Token je vrlo značajan za server čiji servis može provjeriti radi li se o valjanom tokenu. To je jedna od korisnih metodi *Playfab Server API*-a S druge strane, metode za registraciju i prijavu su dio *Playfab Client API-a*. Dakle, unutar našeg servisa kreirali smo klasu *ValidateClient* koja izvršava provjeru valjanosti *SessionTicket-a*

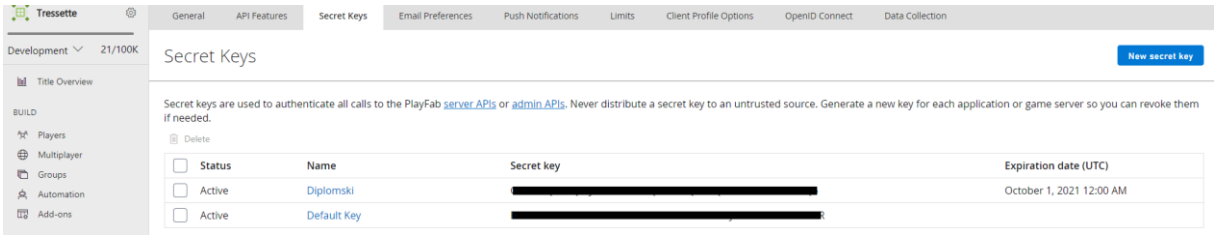
```
17 references
public class ValidateClient
{
    static readonly object _object = new object();
    27 references
    public AuthResponse Response { get; set; }
    8 references
    public ValidateClient(string sessionTicket)
    {
        this.ValidateSessionTicket(sessionTicket).Wait();
    }

    1 reference
    private async Task<AuthResponse> ValidateSessionTicket(string sessionTicket)
    {
        try
        {
            var url = "https://6D881.playfabapi.com/Server/AuthenticateSessionTicket";
            using (var client = new HttpClient())
            using (var request = new HttpRequestMessage(HttpMethod.Post, url))
            {
                Console.WriteLine(sessionTicket);
                AuthenticateSessionTicketPostBody body = new AuthenticateSessionTicketPostBody();
                body.sessionTicket = sessionTicket;
                var json = JsonConvert.SerializeObject(body);
                using (var stringContent = new StringContent(json, Encoding.UTF8, "application/json"))
                {
                    request.Content = stringContent;
                    request.Headers.Add("X-SecretKey", "C[redacted]");
                    using (var response = await client
                        .SendAsync(request, HttpCompletionOption.ResponseHeadersRead)
                        .ConfigureAwait(false))
                    {
                        Monitor.Enter(_object);
                        response.EnsureSuccessStatusCode();
                        HttpContent requestContent = response.Content;
                        var stringResponse = await requestContent.ReadAsStringAsync();
                        AuthResponse res = JsonConvert.DeserializeObject<AuthResponse>(stringResponse);
                        this.Response = res;
                        Monitor.Exit(_object);
                        return res;
                    }
                }
            }
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
            return new AuthResponse();
        }
    }

}
```

Slika 35: ValidateClient klasa za provjeru tokena

Primijetimo da kod kreiranja zahtjeva na *Playfab Server API*, postavljamo *C-SecretKey* atribut u zaglavlje HTTP zahtjeva. Vrijednost ovog atributa treba biti privatni ključ koji možemo generirati unutar samog web sučelja. Osim za *Playfab Server API*, privatni ključevi su potrebni i za *Playfab Admin API.*

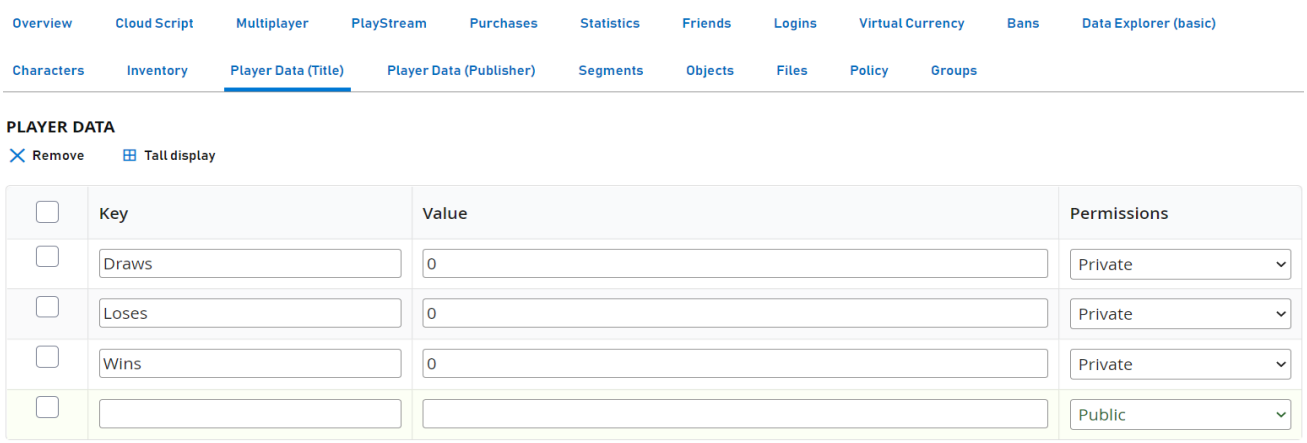


Slika 36: Prikaz tajnih ključeva generiranih unutar Playfab web sučelja

Ukoliko je token valjan, API će servisu vratiti vrijednosti o korisniku, poput njegove Playfab ID vrijednost. Na taj način znamo da je autorizacija korisnika uredno prošla.

4.1.4. Rangiranje igrača

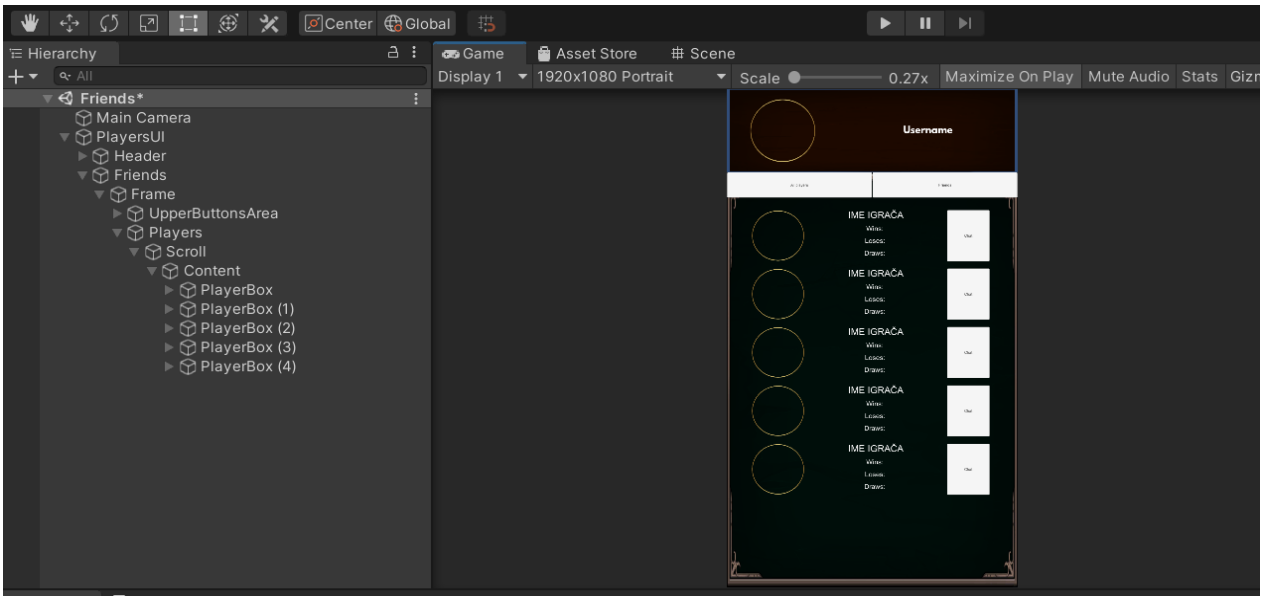
Rangiranje igrača je važno za igre s više igrača iz razloga što su igrači po prirodi kompetitivni, i to daje dodatan draž igri. Mi smo za potrebu rangiranja kreirali tri vrijednosti – pobjede, izgubljene te neriješene partije. Svaki put kada igrač završi sa igrom, ovisno o ishodu mu se obnovi njegova statistika.



Slika 37: Prikaz vrijednosti kreiranih za svakog registriranog igrača

4.1.5. Lista prijatelja

Lista prijatelja je također izrazito važna značajka igre za više igrača. Naime, važno je omogučiti igračima koji se međusobno poznavaju, ili onima koji su se upoznali putem komunikacije unutar igre da se mogu međusobno dodati u listu prijatelja kako bi mogli skupa igrati u budučnosti. Pogledajmo kako izgleda planirani dizajn scene za prikaz svih igrača te liste prijatelja.



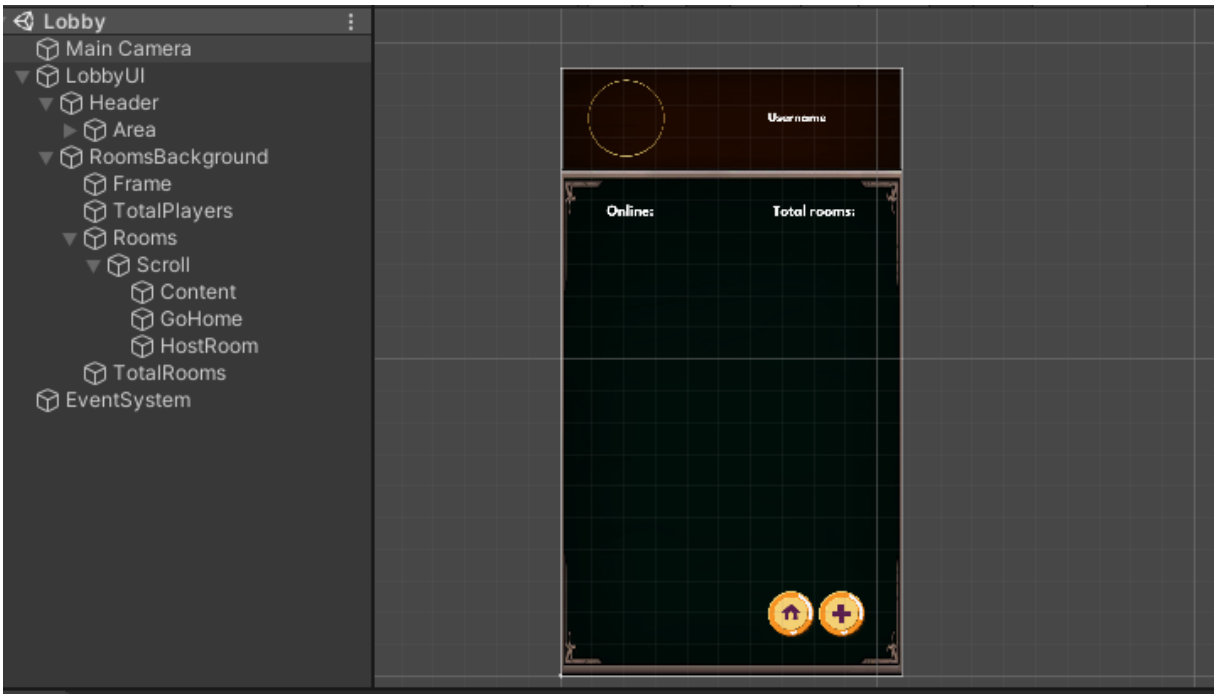
Slika 38: Scena za prikaz liste prijatelja

4.2. Spajanje igrača u zajedničku sobu

Pridruživanje igrača jednih drugima za igranje interaktivne igre nije lagan zadatak. Za uspješnu igru korisničko iskustvo treba biti efikasno i efektivno. Dakle, moramo se pobrinuti i za uredan dizajn te implementirati sve procese koji pospješuju korisnička iskustva.

4.2.1. Predvorje

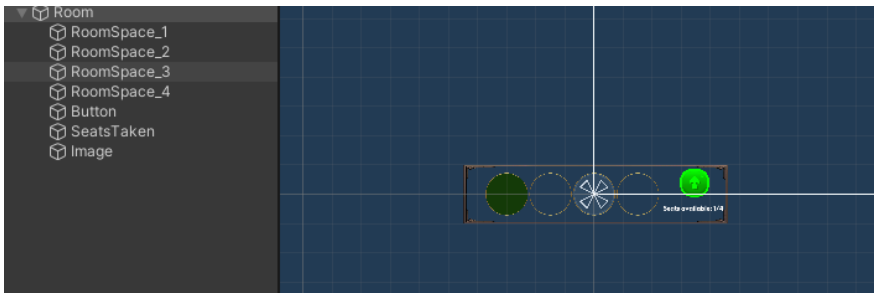
Nakon uspješne prijave, ili registracije. korisnici pristupaju u predvorje. U predvorju se nalaze ostali igrači koji u datom trenutku nisu u igri, ili u nekoj od soba koje čekaju na igru.



Slika 39: Prikaz scene za predvorje

Da dobijemo prikaz sobe, potrebna su nam bila dva panela. Jedan će se nalaziti na vrhu ekrana te će sadržavati avatar sliku korisnika i njegovo korisničko ime. Drugi panel u sebi sadržava *Scroll* UI objekt te mu je dijete objekt s *Grid Layout Group* komponentom, pod nazivom *Content*. Kao takvi, omogućavaju unos ostalih objekata u obliku liste, kao djecu objekta *Content*. Takve ih se također može podizati i spuštati unutar *Scroll* objekta.

Objekte djecu koji idu unutar *Content* objekta smo spremili u projektni folder te ćemo ih programski instancirati u izvođenju. Roditelj tog objekta je Room panel koji ispod sebe sadrži grupu elemenata djece koji predstavljaju jednu sobu unutar predvorja.



Slika 40: Prikaz spremljenog objekta za prikaz sobe unutar predvorja

Cijeli *Room* panel će tako putem svojih *RoomSpace* elemenata na interaktivan način reći koliko se korisnika nalazi u toj sobi. Panel će također korisnika informirati o dostupnosti sobe preko gumba i prikazati broj slobodnih mjesta u sobi.

4.2.2. Igrač kao pridruženi korisnik

Ako se korisnik želi pridružiti nekoj od postojećih soba, to može učiniti iz scene za predvorje. Pri samom pokretanju scene, *LobbyUI* element preko svoje *OnLobbyLoaded* skripte šalje serveru zahtjev sa *YOGIVEMEROOMINFO* šifrom. Unutar JSON stringa se sprema i *SessionTicket* token.

```
TcpKlijent klijent = new TcpKlijent();
klijent.PosaljiServeru("{\"commandId\":\"YOGIVEMEROOMINFO\", \"sessionTicket\":\"" + MyData.Instance.SessionTicket + "\"}");
string odgovor = klijent.PrimiOdServera();
//var sth = JsonConvert.DeserializeObject<object>(odgovor);
Debug.Log("Primio sam odgovor:" + odgovor);
if(!odgovor.Contains("Netocni podaci"))
{
    LobbyResponse response = JsonConvert.DeserializeObject<LobbyResponse>(odgovor);
    DataManager dm = new DataManager();
    dm.SetLobbyRooms(response.Rooms);
}
klijent.ZatvoriSocket();
```

Slika 41: Isječak koda za slanje zahtjeva za pridruživanjem sobi

Nakon što klijent objekt pozove *PosaljiServeru* metodu, čeka se na odgovor. Server u međuvremenu primi zahtjev i obradi ga prema svojoj implementaciji.

```
if (jsonObject.CommandId == "YOGIVEMEROOMINFO")
{
    ValidateClient client = new ValidateClient(jsonObject.SessionTicket);
    if (client.Response != null)
    {
        if (client.Response.code == 200)
        {
            if (Singleton.Instance.GetPlayers().ContainsKey(client.Response.data.UserInfo.PlayFabId))
            {
                string array = "[";
                for (int i = 0; i < Singleton.Instance.GetPlayers().Values.Count; i++)
                {
                    foreach (var item in Singleton.Instance.GetPlayers())
                    {
                        Console.WriteLine("item");
                    }
                    array += JsonConvert.SerializeObject(Singleton.Instance.GetPlayers()) + ",";
                }
                array = array.Remove(array.Length - 1);
                array += "]";
                DataManager dm = new DataManager();
                var players = dm.GetAllPlayers();
                var currentPlayer = players[client.Response.data.UserInfo.PlayFabId];
                string playersJson = JsonConvert.SerializeObject(players.Values, Formatting.Indented);
                var rooms = dm.GetAllRooms();
                string roomsJson = JsonConvert.SerializeObject(rooms);
                json = "{\"NumberOfPlayers\":\"" + players.Values.Count + "\", \"Players\":" + playersJson + ", \"Rooms\": " + roomsJson + "}";
                Console.WriteLine(json);
            }
        }
    }
}
```

Slika 42: Prikaz isječka koda za obradu zahtjeva za pridruživanjem sobe

Ukoliko je prošla autorizacija korisnika, vračaju mu se podaci o predvorju igre. Unutar odgovora može se pronaći podatak o količini igrača koji se nalaze u predvorju, njihovi podaci, kao i podaci o sobama koje su kreirane u vremenu izvršavanja. Potom se na klijentskoj aplikaciji ti primljeni podaci obrađuju i spremaju. Ovisno o broju soba koje postoje u trenutku izvršavanja kreiraju se klonovi *Room* objekta.

```
DataManager dm = new DataManager();
string username = dm.GetMyUsername();
if (username != "" && username != null)
    GameObject.Find("Username").GetComponent<Text>().text = username;
List<Room> rooms = dm.GetLobbyRooms();
if (rooms != null)
{
    foreach (var room in rooms)
    {
        var roomObject = GameObject.Find(room.Id);
        if (roomObject != null)
            Destroy(roomObject);
        GameObject contentUI = GameObject.Find("Content");
        var newRoom = Instantiate(RoomUI, new Vector3(0, 0, 0), Quaternion.identity);
        newRoom.transform.SetParent(contentUI.transform);
        newRoom.name = room.Id;
        newRoom.GetComponent<LobbyRoomUIData>().SetRoomUIData(room);
    }
}
```

Slika 43: Prikaz isječka koda za prikaz soba unutar scene za predvorje

Kako se ovaj isječak koda nalazi u *Update* metodi naslijeđene iz *MonoBehaviour* klase, tako moramo voditi računa o tome da se prije instanciranja novog objekta uništi onaj koji je istanciran u prošloj iteraciji igre. Sa Metodama *Instantiate* i *SetParent* instanciramo objekt te ga smješamo kao dijete *Content* elementa.

Novoj sobi se pridružuje serijski broj pod ime te se poziva metoda SetRoomUIData u kojoj se dodatno mijenjaju elementi kako bi se ispravno prikazali podaci o kreiranim sobama.

```
1 reference
public void SetRoomUIData(Room room)
{
    this.room = room;
    for(int i=0; i<4; i++)
    {
        if(i == 0)
        {
            GameObject seat1 = GameObject.Find("RoomSpace_1");
            if (room.Players.Length > i)
                seat1.GetComponent<Image>().sprite = OccupiedSeat;
            else
                seat1.GetComponent<Image>().sprite = EmptySeat;
        }
        else if(i == 1)
        {
            GameObject seat2 = GameObject.Find("RoomSpace_2");
            if (room.Players.Length > i)
                seat2.GetComponent<Image>().sprite = OccupiedSeat;
            else
                seat2.GetComponent<Image>().sprite = EmptySeat;
        }
        else if (i == 2)
        {
            GameObject seat3 = GameObject.Find("RoomSpace_3");
            if (room.Players.Length > i)
                seat3.GetComponent<Image>().sprite = OccupiedSeat;
            else
                seat3.GetComponent<Image>().sprite = EmptySeat;
        }
        else if (i == 3)
        {
            GameObject seat4 = GameObject.Find("RoomSpace_4");
            if (room.Players.Length > i)
                seat4.GetComponent<Image>().sprite = OccupiedSeat;
            else
                seat4.GetComponent<Image>().sprite = EmptySeat;
        }
    }
    GameObject seatsTaken = GameObject.Find("SeatsTaken");
    seatsTaken.GetComponent<Text>().text = "Seats available: " + (4 - room.Players.Length) + "/4";
```

Slika 44: Isječak koda za prikaz okupiranih i slobodnih sobe unutar scene za predvorje

Primjećujemo da se u ovom isječku koda namještaju vrijednosti za dinamički kreirane sobe. U elementima koji prikazuju sjedeća mjesta se izmjenjuju slike, ovisno o tome je li mjesto slobodno ili zauzeto. Također se i komponenti teksta pridružuje vrijednost koja govori korisniku koliko ima slobodnih mjesta u sobi.

Slika 45: Prikaz sobe u predvorju

Kako dinamički kreirane sobe imaju i gumb, tada mu se unutar *SetRoomUIData* definira slušač koji se izvršava na događaj klika.

```
this.GetComponent<Button>().onClick.AddListener(() =>
{
    TcpKlijent klijent = new TcpKlijent();
    klijent.PosaljiServeru("{\"commandId\":\"YOIWANNAJOIN\", \"sessionTicket\":\"" + MyData.Instance.SessionTicket + "\", \"roomId\": \""+room.Id+"\"}");
    string odgovor = klijent.PrimiOdServera();
    Debug.Log(odgovor);
    JoinLobbyRoomResponse response = JsonConvert.DeserializeObject<JoinLobbyRoomResponse>(odgovor);
    DataManager dm = new DataManager();
    dm.SetRoom(response.MyData);
    SceneManager.LoadScene("Room");
});
```

Slika 46: Isječak koda za osluškivanje gumba za pridruživanje sobi

Primjećujemo da klikom na gumb sobe šalje se serveru *YOIWANNAJOIN* komanda, zajedno sa autorizacijskim tokenom te šifrom te specifične sobe. Ukoliko igrač pošalje valjani token, servis će obnoviti svoje spremljene podatke te vratiti nove podatke o stanju igre u izvođenju.

```
if(jsonObject.CommandId == "YOIWANNAJOIN")
{
    ValidateClient client = new ValidateClient(jsonObject.SessionTicket);
    if(client.Response != null)
    {
        DataManager dm = new DataManager();
        var player = dm.GetOnePlayer(client.Response.data.UserInfo.PlayFabId);
        player.InGameStatus = "ROOM";
        var players = dm.GetAllPlayers();
        players[client.Response.data.UserInfo.PlayFabId] = player;
        var room = dm.GetOneRoom(jsonObject.RoomID);
        if (player != null && room != null)
        {
            dm.AddPlayerToRoom(player, jsonObject.RoomID);
            json = "{\"ResponseId\":\"YO\", \"MyData\":" + JsonConvert.SerializeObject(room) + "}";
        }
        else
            Console.WriteLine("User ne postoji u dictionaryu. ne mogu se spojiti sobi");
    }
}
```

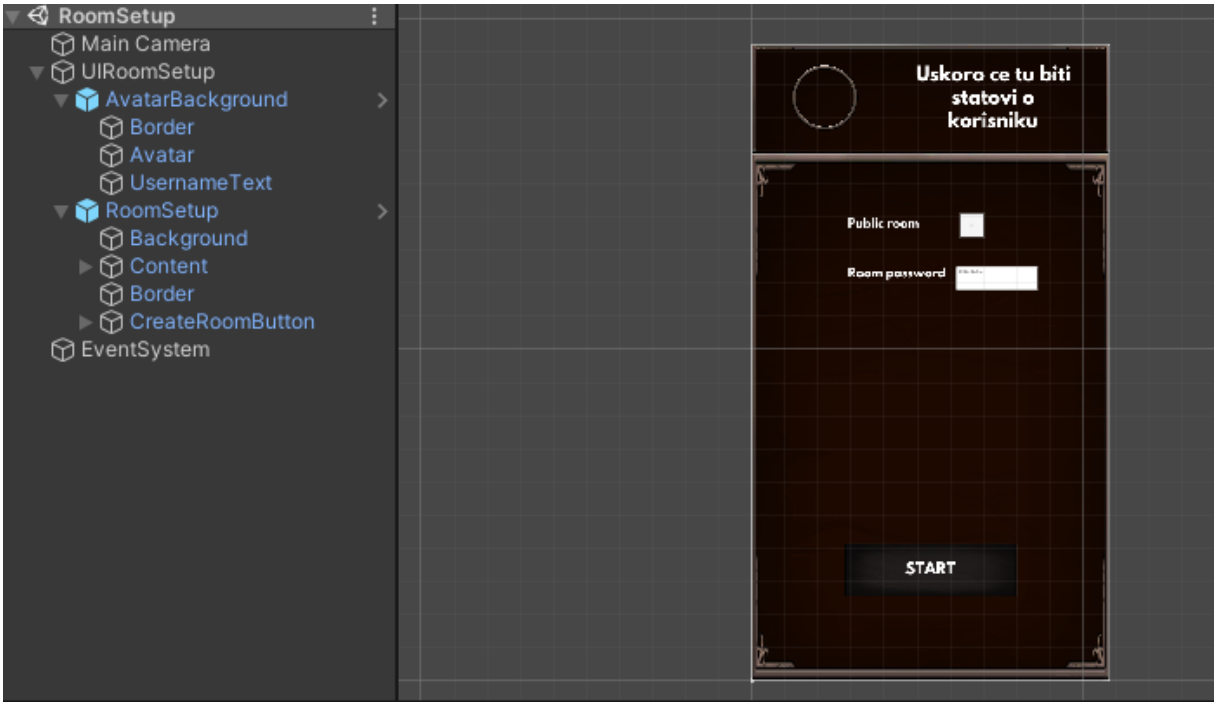
Slika 47: Isječak koda za obradu zahtjeva za pridruživanjem sobi

Kada se prime nazad podaci koji su se obnovili na servisu te ih klijentska aplikacija također obnovi u svojoj radnoj memoriji, poziva se *Room* scena putem *LoadScene* metode.

4.2.3. Igrač kao vlasnik sobe

Ako igrač želi biti vlasnik sobe, mora ju kreirati. Tada klikom na gumb sa sličicom na plus znak otvaramo scenu za kreiranje sobe - *RoomSetup*.

4.2.3.1. Forma za kreiranje sobe



Slika 48: Scena za kreiranje sobe unutar Unity programskog alata

Iako trenutno izgleda jako jednostavno, ova scena je kao takva proširiva sa različitim zahtjevima korisnika te pravilima igre koju korisnici igraju. Naša trenutna scena za kreiranje sobe sadrži *Toggle* objekt kojim će vlasnik sobe odrediti radi li se o javnoj ili privatnoj sobi te odrediti lozinku sobe unutar polja za unos.

Klikom na gumb *Start,* preko mehanizma slušača gumba poziva se metoda *OnCreateRoomButtonClick,* koja će serveru poslati zahtjev za kreiranjem sobe.

```
0 references
public void OnCreateRoomButtonClick()
{
    Debug.Log("Kreiram sobu");
    Debug.Log(PublicRoomToggle.isOn);
    Debug.Log(Password.ToString());
    TcpKlijent client = new TcpKlijent();
    string sessionTicket = new DataManager().GetMySessionTicket();
    if(sessionTicket != null)
    {
        client.PosaljiServeru("{\"commandId\":\"YOIWANNAHOST\", \"sessionTicket\":\"" + sessionTicket + "\"}");
        string odgovor = client.PrimiOdServera();
        Debug.Log(odgovor);
        RoomSetupResponse response = JsonConvert.DeserializeObject<RoomSetupResponse>(odgovor);
        DataManager dm = new DataManager();
        dm.SetRoom(response.MyData);
    }
    else
    {
        Debug.Log("No session ticket :(");
    }
    SceneManager.LoadScene("Room");
}
```

Slika 49: Isječak koda za slanje zahtjeva za kreiranje sobe

Slanjem poruke serveru sa šifrom *YOIWANNAHOST* i slanjem tokena pronađenog unutar *SessionTicket-a*, server preko svog servisa izvršava obradu tog zahtjeva.

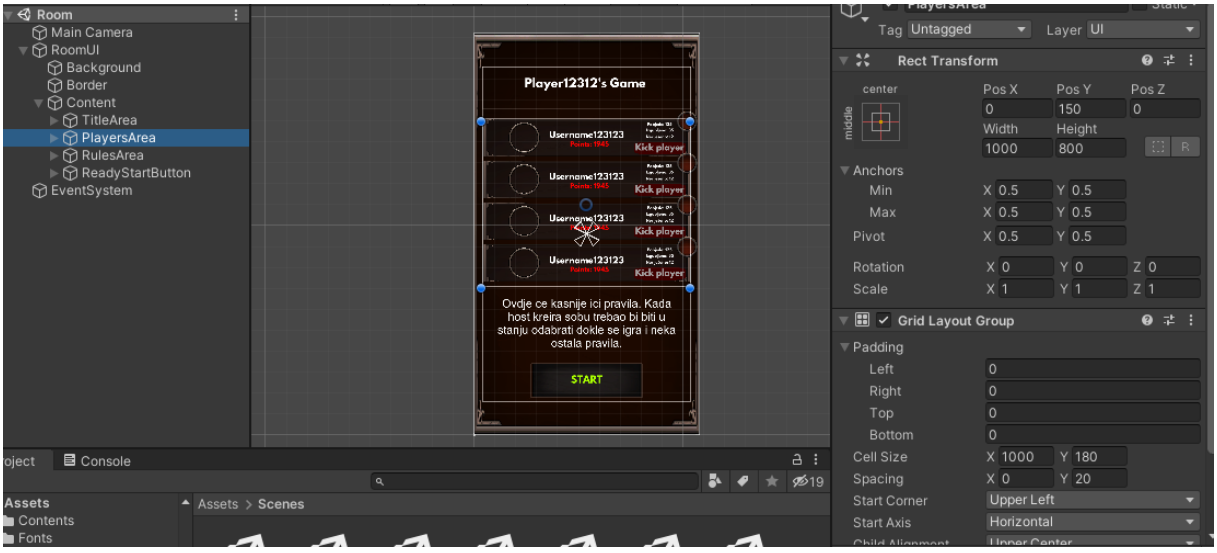
```
if (jsonObject.CommandId == "YOIWANNAHOST")
{
    ValidateClient client = new ValidateClient(jsonObject.SessionTicket);
    if (client.Response != null)
    {
        Room room = new Room();
        room.HostPlayfabId = client.Response.data.UserInfo.PlayFabId;
        room.Id = Guid.NewGuid().ToString();
        var players = new DataManager().GetAllPlayers();
        Player[] roomPlayers = new Player[1];
        var player = players[client.Response.data.UserInfo.PlayFabId];
        player.InGameStatus = "ROOM";
        players[client.Response.data.UserInfo.PlayFabId] = player;
        roomPlayers[0] = players[client.Response.data.UserInfo.PlayFabId];
        room.Players = roomPlayers;
        json = "{\"ResponseId\":\"YO\", \"MyData\":" + JsonConvert.SerializeObject(room) + "}";
        Singleton.Instance.AddRoom(room);
    }
}
```

Slika 50: Isječak koda za obradu zahtjeva za kreiranjem sobe

Iz bloka programa koji obrađuje zahtjev za kreiranjem sobe vidimo kako servis kreira novu sobu u realnom vremenu te joj pridružuje nanovo generiranu *Guid* vrijednost kao šifru sobe. Potom unosi igrača koji je poslao zahtjev kao vlasnika sobe i njenog jedinog trenutnog člana. Naposljetku korisniku vraća JSON poruku o stanju unutar sobe.

Kada je korisnik primio nazad podatke o sobi koju je kreirao, iz predzadnje slike vidimo da se poziva naposljetku poziva *LoadScene* metoda iz *SceneManager* objekta te se tipa pokreće nova scena za sobu.

4.3. Zajednička soba



Slika 51: Prikaz scene za zajedničku sobu unutar programskog alata Unity

Korištenjem roditelja panela te grupiranih objekata korisničkog iskustva, napravljen je i korisnički prikaz sobe. U ovoj sceni imamo *Content* objekt koji kao *Panel* objekt grupira područje za naziv sobe, područje koje prikazuje status igrača unutar sobe, te područje namijenjeno za pravila sobe i/ili igre, kao i gumb.

4.3.1. Prikazivanje ostalih igrača

Kao članovi sobe, igrači moraju u realnom vremenu imati informaciju o stanju u sobi. U bilo kojem trenutku se netko može priključiti u sobu, ili može izaći. Ostali igrači moraju biti obaviješteni o takvim događajima. Korisničke aplikacije tako radi toga unutar *Update* metode šalje *YOIMINROOM* zahtjev servisu. Unutar JSON stringa šalje se i autorizacijski token, serijski broj sobe te serijski broj pošiljatelja zahtjeva. Ukoliko se radi o zahtjevu s valjanim autorizacijskim tokenom, u odgovoru će dobiti podatke potrebne za prikaz elemenata.

```
TcpKlijent klijent = new TcpKlijent();
klijent.PosaljiServeru("{\"CommandId\":\"YOIMINROOM\", \"SessionTicket\":\"" + MyD
string odgovor = klijent.PrimiOdServera();
Debug.Log(odgovor);
if(!odgovor.Contains("Netocni podaci"))
{
    RoomUIResponse response = JsonConvert.DeserializeObject<RoomUIResponse>(odgovc
    DataManager dm = new DataManager();
    if (response != null)
    {
        if (response.MyData != null)
        {
            if (this.RoomDataChanged(response.MyData))
            {
                dm.SetRoom(response.MyData);
                this.roomUpdated = true;
            }
        }
    }
}
```

Slika 52: Isječak koda za slanje zahtjeva o stanju unutar sobe

Za svakog igrača se potom putem spremljenih podataka unutar Update metode prikazuju ili isključuju elementi sjedećih mjesta, ovisno o tome jesu li okupirani s nekim igračem.

```
for(int i=0; i<this.room.Players.Length; i++)
{
    if(i == 0)
    {
        if(this.playerBox1 != null)
        {
            if (!this.playerBox1.activeSelf)
                this.playerBox1.SetActive(true);
            if (this.room.Players[i].InGameStatus == "READY")
                this.playerStatus1.GetComponent<Image>().sprite = this.PlayerReady;
            else
                this.playerStatus1.GetComponent<Image>().sprite = this.PlayerNotReady;
            if (room.Players.Length == i + 1)
                DisableEmptyBoxesIfEnabled(room.Players.Length);
        }
    }
    else if(i == 1)
    {
        if (!this.playerBox2.activeSelf)
            this.playerBox2.SetActive(true);
        if (this.room.Players[i].InGameStatus == "READY")
            this.playerStatus2.GetComponent<Image>().sprite = this.PlayerReady;
        else
            this.playerStatus2.GetComponent<Image>().sprite = this.PlayerNotReady;
        if (room.Players.Length == i + 1)
```

Slika 53: Isječak koda za prikaz okupiranog mjesta unutar sobe

Kada korisnik uđe u sobu stvori se novo mjesto unutar sobe. Ukoliko isti igrač izađe, tada će se njegovo mjesto ukloniti . Pogledajmo kako izgleda finalno forma za prikaz zajedničke sobe.

Slika 54: Forma za zajedničku sobu

4.3.2. Izbacivanje igrača

Igrač kao vlasnik sobe može izbaciti igrača iz sobe. Izbacivanje igrača može izvršiti klikom na gumb s tekstom *Kick player.* Pritom će se izvršiti OnKickPlayerButtonClick() metoda koja je referencirana na objekt gumba.

Slika 55: OnKickPlayerButtonClick metoda

Kod izvršavanja metode, servisu se šalje *YOIMKICKIN* komanda te se unutra JSON stringa prosljeđuje autorizacijski token, serijski broj sobe, serijski broj Playfab igrača koji šalje zahtjev te serijski broj Playfab igrača kojeg se izbacuje iz sobe. Nakon uspješne autorizacije, servis obrađuje *YOIMKICKIN* komandu tako da iz spremljenog stanja soba izbaci željenog igrača te vrati novonastalo stanje

```
if(jsonObject.CommandId == "YOIMKICKIN")
{
    ValidateClient client = new ValidateClient(jsonObject.SessionTicket);
    if(client.Response != null)
    {
        DataManager dm = new DataManager();
        string statusCode = dm.RemovePlayerFromRoom(jsonObject.RoomID, jsonObject.PlayfabId, jsonObject.KickedPlayerId);
        var room = dm.GetOneRoom(jsonObject.RoomID);
        json = "{\"ResponseId\":\""+statusCode+"\", \"MyData\":" + JsonConvert.SerializeObject(room) + "}";
    }
}
```

Slika 56: Isječak koda za obradu zahtjeva za izbačenjem igrača iz sobe

Kada se korisniku vrate uspješno podaci, zbog *Update* metode se igrač ukloni iz sobe na način da se sakriju svi UI elementi koji su ga reprezentirali. To izvršava unutar *DisableEmptyBoxesIfEnabled* metode koja isključuje elemente na sceni kod kojih više nema igrača.

```
4 references
public void DisableEmptyBoxesIfEnabled(int numberOfPlayers)
{
    if(numberOfPlayers == 1)
    {
        if (this.playerBox2.activeSelf)
            this.playerBox2.SetActive(false);
        if (this.playerBox3.activeSelf)
            this.playerBox3.SetActive(false);
        if (this.playerBox4.activeSelf)
            this.playerBox4.SetActive(false);
    }
    else if(numberOfPlayers == 2)
    {
        if (this.playerBox3.activeSelf)
            this.playerBox3.SetActive(false);
        if (this.playerBox4.activeSelf)
            this.playerBox4.SetActive(false);
    }
    else if(numberOfPlayers == 3)
    {
        if (this.playerBox4.activeSelf)
            this.playerBox4.SetActive(false);
    }
}
```

Slika 57: DisableEmptyBoxesIfEnabled metoda

4.3.3. Označavanje pridruženog igrača kao spreman

Pridruženi igrač, dakle ne vlasnik sobe, mora informirati druge da je spreman za igru. Drugi igrači prepoznavaju spremnost igrača pomoću okrugle slike sa znakom X.

Slika 58: Prikaz spremnosti igrača unutar igre

Ukoliko igrač označi da je spreman, tada mu se interaktivno promijeni slika spremnosti koja sadržava simbol kvačice. Pogledajmo kako to izgleda na idućoj slici.

Slika 59: Igrač je označio da je spreman za igru

Ovu značajku smo implementirali na način da smo prvo sa klijentske aplikacije na klik gumba serveru poslali zahtjev o obradi spremnosti igrača.

```
TcpKlijent klijent = new TcpKlijent();
klijent.PosaljiServeru("{\"commandId\":\"YOIMREADY\", \"sessionTicket\":\"" + MyData.Instance.SessionTicket
string odgovor = klijent.PrimiOdServera();
RoomUIResponse response = JsonConvert.DeserializeObject<RoomUIResponse>(odgovor);
this.room = response.MyData;
dm.SetRoom(this.room);
```

Slika 60: Isječak koda za slanje zahtjeva o spremnosti igrača

Kada je servis primio zahtjev sa šifrom *YOIMREADY,* započinje obrada zahtjeva. Ukoliko je korisnik uspješno autoriziran, servis će obnoviti podatke o njemu tako da mu se promijeni status unutar sobe u *READY.*

```
if (jsonObject.CommandId == "YOIMREADY")
{
    var token = new TokenManager().DecryptToken(jsonObject.Jwt);
    if (token != null)
    {
        DataManager dm = new DataManager();
        dm.UpdatePlayerIngameStatus(jsonObject.PlayfabId, jsonObject.RoomID, "READY");
        var room = dm.GetOneRoom(jsonObject.RoomID);
        json = "{\"ResponseId\":\"OK\", \"MyData\":" + JsonConvert.SerializeObject(room) + "}";
    }
}
```

Slika 61: Isječak koda za obradu zahtjeva o spremnosti igrača

4.3.4. Pokretanje igre od strane vlasnika sobe

Kada su svi igrači koji su se pridružili sobi spremni za igru, tada vlasnik sobe može istu i započeti. Implementacija ovoga je slična i prethodnoj implementaciji, kada smo obradili spremnost igrača.

Prvo smo poslali zahtjev za pokretanjem igre. Sa klijenta se šalje JSON string sa YOSTARTGAME šifrom te ostalim podacima potrebnim za obradu zahtjeva.

```
TcpKlijent klijent = new TcpKlijent();
klijent.PosaljiServeru("{\"commandId\":\"YOSTARTGAME\", \"sessionTicket\":\"" + MyData.Instance.SessionTicket + "\", \"playfabId\":\"" + new
string odgovor = klijent.PrimiOdServera();
RoomUIResponse response = JsonConvert.DeserializeObject<RoomUIResponse>(odgovor);
if (response.ResponseId == "OK")
{
    this.room = response.MyData;
    dm.SetRoom(this.room);
}
```

Slika 62: Isječak koda za zahtjev vlasnika sobe za pokretanjem igre

Kada servis primi zahtjev, prvo obrađuje ispravnost tokena te ukoliko je sve prošlo kako treba, započinje proceduru koja mijenja statuse korisnika iz *READY* u *GAME*. Ukoliko nisu svi pridruženi igrači spremni tada će servis vratiti odgovor sa šifrom *FORBIDDEN*. S druge strane, ako su zaista svi igrači spremni, tada će se vratiti odgovor sa šifrom *OK*.

```
if (jsonObject.CommandId == "YOSTARTGAME")
{
    var token = new TokenManager().DecryptToken(jsonObject.Jwt);
    if (token != null)
    {
        DataManager dm = new DataManager();
        var players = dm.GetAllPlayers();
        var rooms = dm.GetAllRooms();
        var room = dm.GetOneRoom(jsonObject.RoomID);
        bool readyToStart = true;
        for (int i = 1; i < room.Players.Length; i++)
        {
            if (room.Players[i].InGameStatus != "READY")
                readyToStart = false;
        }
        var statusCode = "";
        if (readyToStart)
        {
            for (int i = 0; i < room.Players.Length; i++)
            {
                room.Players[i].InGameStatus = "GAME";
                var player = players[room.Players[i].PlayFabId];
                player.InGameStatus = "GAME";
                players[player.PlayFabId] = player;
            }
            dm.SetAllPlayers(players);
            for (int i = 0; i < rooms.Count; i++)
            {
                if (rooms[i].Id == room.Id)
                    rooms[i] = room;
            }
            dm.SetAllRooms(rooms);
            statusCode = "OK";
        }
        else
            statusCode = "FORBIDDEN";
        json = "{\"ResponseId\":\"" + statusCode + "\", \"MyData\":" + JsonConvert.SerializeObject(room) + "}";
    }
}
```

Slika 63: Isječak koda za obradu zahtjeva za pokretanjem igre

U sljedećim okvirima će se pri zahtjevu za informacijama o sobi zaključiti da svi igrači imaju promjenjeni status te će svi zbog toga pokrenuti scenu za igru.

```
bool playersInGame = true;
for (int i = 0; i < this.room.Players.Length; i++)
{
    if (this.room.Players[i].InGameStatus != "GAME")
        playersInGame = false;
}
if (playersInGame)
    SceneManager.LoadScene("GAME");
```

Slika 64: Isječak koda za pristupanje sceni za igru

4.4. Igranje igre

Kada su igrači ušli u igru, započinje mnoštvo novih procesa koji se moraju obraditi kako bi igrači imali visoko kvalitetno korisničko iskustvo te interaktivnost. U ovom podpoglavlju ćemo obraditi procese obrade početka igre te njihovih igračih aktivnosti. S obzirom da razvijamo kartašku igru, neki procesi će biti vezani uz rad s kartama.

4.4.1. Obrada početka igre

Na početku igre, u metodi *Start* vlasnik igre šalje zahtjev za obradom kreirane igre serveru. Dovoljno je da jedan igrač obznani serveru da je igra uspješno krenula pa će to raditi prethodni vlasnik sobe.

```
DataManager dm = new DataManager();
Room room = dm.GetRoom();
if(dm.GetMyPlayfabId() == room.HostPlayfabId)
{
    TcpKlijent klijent = new TcpKlijent();
    klijent.PosaljiServeru("{\"commandId\":\"YOGAMESTARTED\", \"sessionTicket\":\"" + MyData.Instance.Sessi
    string odgovor = klijent.PrimiOdServera();
    GameResponse response = JsonConvert.DeserializeObject<GameResponse>(odgovor);
    this.game = response.MyData;
    dm.SetMyGame(this.game);
}
```

Slika 65: Isječak koda za zahtjev o uspješno započetoj igri

Servis kada primi zahtjev sa šifrom YOGAMESTARTED, nakon uspješne autorizacije korisnika kreira novi objekt igre te unosi igrače iz sobe u njega. Također pomoću generatora slučajnih brojeva definira prvog igrača koji miješa karte.

```
if (jsonObject.CommandId == "YOGAMESTARTED")
{
    var token = new TokenManager().DecryptToken(jsonObject.Jwt);
    if (token != null)
    {
        DataManager dm = new DataManager();
        var room = dm.GetOneRoom(jsonObject.RoomID);
        dm.SetAllPlayersIngameStatus(room);
        var game = new Game();
        game.Players = room.Players;
        var rand = new Random();
        var dealer = rand.Next(0, 3);
        game.DealerPlayfabId = game.Players[dealer].PlayFabId;
        game.GameId = Guid.NewGuid();
        game.Status = "STARTED";
        game.RoomId = room.Id;
        dm.AddOneGameToGames(game);
        var statusCode = "OK";
        json = "{\"ResponseId\":\"" + statusCode + "\", \"MyData\":" + JsonConvert.SerializeObject(game) + "}";
    }
}
```

Slika 66: Isječak koda za obradu pokrenute igre

Svi igrači će od servera tokom svakog okvira u sekundi zatražiti informacije o stanju igre. Tp će napraviti sa slanjem zahtjeva pod šifrom *YOGIVEMEGAMEINFO*.

```
TcpKlijent klijent = new TcpKlijent();
klijent.PosaljiServeru("{\"commandId\":\"YOGIVEMEGAMEINFO\", \"sessionTicket\":\"" + MyData.In
string odgovor = klijent.PrimiOdServera();
GameResponse response = JsonConvert.DeserializeObject<GameResponse>(odgovor);
```

Slika 67: Slanje zahtjeva o stanju igre

Potom će server, opet ovisno o uspješnoj autorizaciji, obraditi njihov zahtjev i vratiti im sve informacije o stanju igre u kojoj se nalaze.

```
if (jsonObject.CommandId == "YOGIVEMEGAMEINFO")
{
    var token = new TokenManager().DecryptToken(jsonObject.Jwt);
    if (token != null)
    {
        DataManager dm = new DataManager();
        Game game = dm.GetOneGame(jsonObject.GameId);
        var statusCode = "OK";
        json = "{\"ResponseId\":\"" + statusCode + "\", \"MyData\":" + JsonConvert.SerializeObject(game) + "}";
    }
}
```

Slika 68: Obrada zahtjeva o stanju igre

4.4.2. Dijeljenje karata

Kako igrač svaki okvir u sekundi dobije točno stanje o igri, tako može znati je li je njega red za dijeljenje karata. Ukoliko je, tada on iste igrače promiješa te prosljedi serveru promiješani špil karata. Prije slanja zahtjeva prvo se poziva metoda *DealCards*.

```
1 reference
public CardData[] DealCards(Game game)
{
    bool cardsDealt = false;
    if (cardsDealt)
    {
        var cards = new CardData[40];
        for(int i=1; i<=40; i++)
        {
            CardData card = new CardData();
            card.Value = i.ToString();
            cards[i - 1] = card;
        }
        RNGCryptoServiceProvider random = new RNGCryptoServiceProvider();
        cards = cards.OrderBy(x => Next(random)).ToArray();
        return cards;
    }
    else
    {
        Debug.Log("Karte nisu podijeljene iz nekog cudnog razloga. NE mogu nastaviti sa igrom.");
        return null;
    }
}
```

Slika 69: Metoda DealCards

Nakon što su karte promiješane, šaljemo serveru zahtjev sa šifrom YOISHUFFLED te ostalim podacima potrebnim za obradu ovog zahtjeva.

```
var cards = this.GetComponent<CardManager>().DealCards(response.MyData);
this.game.ShuffledCards = cards;
dm.SetMyGame(game);
klijent.PosaljiServeru("{\"commandId\":\"YOISHUFFLED\", \"sessionTicket\":\"" + MyData.Instance.Sessi
odgovor = klijent.PrimiOdServera();
response = JsonConvert.DeserializeObject<GameResponse>(odgovor);
if(response != null)
{
    if(response.MyData != null)
        this.game = response.MyData;
}
```

Slika 69: Isječak koda za zahtjev za obradom pomiješanih karata

I ovdje servis obrađuje zahtjev tako da prvo autorizira korisnika te potom podijeli špil svim igralima. Tako podijeljenje karte se spremaju u objekte korisnika koji se nalaze unutar sobe te se vračaju korisnicima svaki put kada korisnici zatraže informacije o igri pomoću zahtjeva sa YOGIVEMEGAMEINFO šifrom.

4.4.3. Igranje karte

Svaka instancirana karta ima svoj slušač za klik koji označava da igrač baca željenu kartu. Kada se okine događaj, korisnik šalje zahtjev serveru za obradom ove akcije. U skripti imamo i definiranu zastavicu *CanPlay* koja je postavljena kao istina ukoliko je igrača zaista red igrati. Dakle, zahtjev će se poslati samo kada je korisnika zaista red igrati.

```
card.GetComponent<Button>().onClick.AddListener(() =>
{
    if (this.CanPlay)
    {
        card.GetComponent<RectTransform>().sizeDelta = new Vector2(100, 300);
        card.transform.SetParent(this.Table.transform);
        TcpKlijent klijent = new TcpKlijent();
        klijent.PosaljiServeru("{\"commandId\":\"YOIPLAYEDCARD\", \"sessionTicket\":\"" + MyData.Instance.SessionTicket + "\", \"Jv
        string odgovor = klijent.PrimiOdServera();
        GameResponse response = JsonConvert.DeserializeObject<GameResponse>(odgovor);
        DataManager dm = new DataManager();
        dm.SetMyGame(response.MyData);
        this.CanPlay = false;
        for(int i=0; i<response.MyData.Players.Length; i++) {
            if(response.MyData.Players[i].PlayfabId == dm.GetMyPlayfabId())
            {
                if (response.MyData.Players[i].Hand.Length == 0)
                    dm.SetMyHandEmpty(true);
            }
        }
    }
});
```

Slika 70: Isječak koda za slanje zahtjeva o odigranoj karti

Pogledajmo kako izgleda obrada zahtjeva na servisu. Prvo se autorizira korisnika te se potom obrade podaci tako da se osvježi stanje ruke koju korisnik ima. Dakle, ukloni se karta iz njegove ruke. Također se ista karta doda u polje koje označava karte koje se trenutno nalaze na stolu. To je važno je podaci moraju biti dostupni igračima kada budu zahtjevali stanje o igri.

```
if (jsonObject.CommandId == "YOIPLAYEDCARD")
{
    var token = new TokenManager().DecryptToken(jsonObject.Jwt);
    if (token != null)
    {
        DataManager dm = new DataManager();
        var games = dm.GetAllGames();
        var game = games[jsonObject.GameId];
        var players = dm.GetAllPlayers();
        var player = players[token.Id];
        var newHand = player.Hand.Where(val => val.Value != jsonObject.CardValue).ToArray();
        player.Hand = newHand;
        players[token.Id] = player;
        var index = -1;
        for (int i = 0; i < game.Players.Length; i++)
        {
            if (game.Players[i].PlayFabId == game.DealerPlayfabId)
                index = i;
        }
        if (index != -1)
        {
            game.Players[index] = player;
            if (index != 4)
                game.PlayerIdToPlay = game.Players[index + 1].PlayFabId;
            else
                game.PlayerIdToPlay = game.Players[0].PlayFabId;
        }
        games[jsonObject.GameId] = game;
        dm.SetAllPlayers(players);
        dm.SetAllGames(games);
        var statusCode = "OK";
        json = "{\"ResponseId\":\"" + statusCode + "\", \"MyData\":" + JsonConvert.SerializeObject(game) + "}";
    }
}
```

Slika 71: Isječak koda za obradu zahtjeva o odigranoj karti

5. Ostali resursi korišteni za razvoj igre za više igrača

Osim samog Unitya, koristili smo i mnoštvo alata i resursa koji su nam omogučili brži i jednostavniji razvoj projekta. Ti resursi su uređivač programskog koda, sustav za verzioniranje, korisni Unity dodaci za razvoj igara za više igrača te ostali korisni alati.

5.1. Uređivač programskog koda

Skripte koje ćemo smo programirali u projektu su pisane u jeziku C# i u klijentskim aplikacijama (Unityu), kao i na pozadinskim aplikacijama (backend). Prema tome, odlučili smo se za korištenje Visual Studio uređivač koda specijalno za C#.

5.2. Sustav za verzioniranje

Verzioniranje je iznimno važno tokom razvoja, kako bi se sačuvali dijelovi koda koji su već napravljeni i testirani. Za verzioniranje sustava u projektu ćemo koristiti Gihub repozitorij te Git Shell aplikaciju.

5.3. Korisni alati za razvoj igara za više igrača u Unityu

5.3.1. Mirror

Testirali smo Mirror alat za razvoj igrača unutar Unitya. Pomoću Mirrora možemo bez ikakvih poteškoća definirati projekt kao klijent ili server, u izvođenju. Kada je projekt u izvođenju definiran kao server, tada klijenti mogu kontaktirati server sa *[TargetRPC]* atributom. Naime, sve što je potrebno, je iznad metode dodati ovaj atribut. S druge strane, *[Command]* atribut se koristi za slanje poruka od servera do klijenta. Opisnik metodi se također dodaje iznad definicije metode.

Primječujemo koliko nam ovaj alat koristi u razvoju. Naime, bez dodatnog implementiranja možemo na jednostavan način spojiti igrače u mrežu, gdje je pritom jedan od projekata u izvođenju server, a ostali su klijenti. Razmjena poruka na mreži je također vrlo jednostavna te ovo ćini Mirror kao odličan izbor za razvoj manjeg projekta. Ipak, mi smo se odlučili za vlastitu implementaciju na mreži kako bi imali veću proširivost projekta te skalabilnost.

5.3.2. ParrelSync

ParrelSync je dodatak za Unity koji nam daje mogučnost pokretanja klonova Unity projekta na kojem radimo. Naime, to je jako korisno za testiranje igre za više igrača. Kako se

radi o više igrača, tada moramo i u isto vrijeme testirati igru za više igrača. Prema tome, ako testiramo lokalno, trebamo pokrenuti više projekata u istom trenutku. ParrelSync nam ovaj proces ubrzava sa svojim upravljačem koji nam daje da pokrenemo klonove glavnog projekta.

Druga važna značajka ParrelSynca je ta da ako promijenio nešto unutar jednog od projekata, tada će se sve izmjene pri spremanju sinkronizirati i na projekte klonove.

5.3.3. Podatkovni sloj

Svaka video igra ima podatke koji moraju biti spremljeni negdje. Neki proizvođači se odlučuju za rješenja trećih strana u oblaku. S druge strane, uglavnom svi proizvođači imaju i svoju bazu podataka gdje spremaju podatke koji su strateški važni za poslovanje i ispravan rad igre.

5.3.4. Ostali korisni izvori

5.3.4.1. Pretvarač JSON stringa u objekt

Tokom razvoja ovog projekta bilo je izrazito korisno koristiti pretvarač JSON stringa u objekt. Takav pretvoreni objekt smo mogli samo prekopirati u projekt, bez da pišemo klase ručno. Jedan od pretvarača može se pronaći na sljedećoj web lokaciji: https://json2csharp.com.

5.3.4.2. OpenGameArt

OpenGameArt je web stranica na kojoj se besplatno mogu preuzimati 2D umjetnosti, 3D umjetnosti, teksture i glazba za igru koju razvijamo.

5.3.4.3. GameArt2d

Valja spomenuti i GameArt2d na kojoj nije toliko široka ponuda, ali besplatni resursi su bitno veće kvalitete nego kakve možemo pronaći na OpenGameArt-u.

5.3.4.4. Unity trgovina dodataka

Iako se većina dodataka u trgovini naplaćuje, postoji mnoštvo besplatnih dodataka različite namjene koji su korisni za preuzimanje te razvoj video igara.

6. Zaključak

U ovom radu bavili smo se izradom igre za više igrača u Unity programskom alatu. Naučili smo što su to mrežni protokoli na kojima se zasniva komunikacija između igrača te kako se implementira transportni sloj za igru.
Uočili smo da je za igru visoke kvalitete potreban server koji će obrađivati sve zahtjeve korisnika. Mi smo u praktičnom dijelu projekta koristili C# za izradu servisa. Sam razvoj može biti jako kompliciran te je jako važno voditi računa o čitkom kodu te pravilima struke o razvoju video igara.

Najveći fokus smo stavili na programski alat Unity koji je specijalizirani sustav za izradu video igara. Sustav je napredan te je potrebno prijašnje iskustvo u programiranju kako bi se lakše naučio koristiti ovaj sustav. Također je korisno prethodno iskustvo u korištenju ostalih alata, posebice alata za uređivanje slika ili videa.

Igre za više igrača su prije svega jako zanimljive. Sve aktivnosti se događaju u realnom vremenu kako bi igre bile interaktivne. Kada se razvija igra za više igrača, treba ju i testirati kao da ju igra više igrača, što zna biti itekako velik problem. Ipak, Unity sadrži kvalitetnu dokumentaciju koja nam je uvelike pomogla u učenju te razvoju.

Obrana ovog rada označava završetak diplomskog studija te stjecanje diplome magistra informatike. Smatram da je zahtjevnost ovog rada zaista bila na razini titule te sam ponosan što sam radio na ovoj temi.

Za kraj se zahvaljujem mentoru, doc. dr. sc. Mladenu Koneckom koji mi je dao svo povjerenje te slobodu kod izrade ovog rada. Također bih se zahvalio kompaniji Tau On-Line koja mi je omogučila da dio radnog vremena uložim u razvoj projekta te pisanje ovog rada.

7. Popis literature

[1] Vinton G. Cerf; Robert E. Kahn (1974.) *A protocol for Packet Network Intercommunication (PDF).* IEEE Transactions on Communications. 22 (5): 637-648.

[2] Wikipedia (2021) *Transmission Control Protocol*, Preuzeto 19. rujna 2021. sa lokacije Transmission Control Protocol - Wikipedia

[3] Wikipedia (2021) *Unity (game engine)* Preuzeto 19. rujna 2021. sa lokacije Unity (game engine) - Wikipedia

[4] Unity Docs (202?) *Scenes* Preuzeto 19. rujna 2021. sa lokacije Unity - Manual: Scenes (unity3d.com)

[5] Unity Docs (202?) *Scripting* Preuzeto 19. rujna 2021. sa lokacije Unity - Manual: Scripting (unity3d.com)

[6] Unity Docs (202?) *RectTransform* Preuzeto 19. rujna 2021. sa lokacije Unity - Scripting API: RectTransform (unity3d.com)

[7] Microsoft (2020) *What is Playfab?* Preuzeto 19. rujna 2021. sa lokacije What is PlayFab? - PlayFab | Microsoft Docs

[8] Shrine (2020) *Multiplayer Game Architecture in Unity* Preuzeto 19. rujna 2021. sa lokacije (1) Multiplayer Game Architecture in Unity - YouTube

[9] OpenPR (2020) *Browser Games Market Report 2020 With Industry Positioning Of Key Vendors And COVID-19 Impact Analysis: Sega, Sony Corporation, Peak Games* Preuzeto 19. rujna 2021. sa lokacije Browser Games Market Report 2020 With Industry Positioning (openpr.com)

8. Popis slika